



Tecnicatura Universitaria en Programación a Distancia

Base de Datos I

Coord. Oscar Londero
Prof. Sergio Neira (Comisión 3)
Prof. Cinthia Rigoni (Comisión 10)
Prof. Gustavo Sturtz (Comisión 14)

Trabajo Final Integrador

(Grupo 10: Empleado - Legajo)

Alumnos:

Agustín Emiliano Sotelo Carmelich (agustinemiliano22@gmail.com) - Comisión 10
Bruno Giuliano Vapore (brunogvapore@gmail.com) - Comisión 10
Santiago Octavio Varela (santiago.varela@tupad.utn.edu.ar) - Comisión 14
Diego Alejandro Velardes (velardesdiego@gmail.com) - Comisión 3

Fecha de Entrega: 23 de Octubre de 2025

Índice

Resumen Ejecutivo.....	2
Reglas de Negocio.....	2
Etapa 1 - Modelado y Definición de Constraints.....	3
Etapa 2 – Generación y carga de datos masivos con SQL puro.....	7
Etapa 3 – Consultas Avanzadas y Reportes.....	13
Etapa 4 – Seguridad e Integridad.....	23
Etapa 5 – Conurrencia y Transacciones.....	33
Anexo I: Uso de IA.....	44
Etapa 1 – Modelado y Definición de Constraints.....	44
Etapa 2 – Generación y carga de datos masivos con SQL puro.....	51
Etapa 3 – Consultas Avanzadas y Reportes.....	60
Etapa 4 – Seguridad e Integridad.....	68
Etapa 5 – Conurrencia y Transacciones.....	90
ANEXO II: Links a repositorio GitHub y Video.....	99

Resumen Ejecutivo

Este trabajo integrador aplica los conocimientos de la materia mediante la creación de una base de datos MySQL para una aplicación Java que maneja una relación uno a uno entre Empleados y Legajos. Buscando simular un entorno real, el proyecto se desarrolló en cinco etapas: modelado e implementación, carga masiva de datos, consultas avanzadas, seguridad y concurrencia. También se contó con el apoyo pedagógico de la IA durante el proceso.

Reglas de Negocio

El sistema de gestión de personal se fundamenta en un conjunto de reglas de negocio diseñadas para garantizar la integridad y coherencia de la información. La correcta identificación del personal es lo más importante, por lo que cada empleado debe registrarse con nombre, apellido y DNI, que son campos definidos como obligatorios para asegurar un registro de datos mínimo y consistente. El DNI funciona como identificador único e irrepetible en toda la organización. Se contempla un campo para el email que, si bien es opcional, debe cumplir con un formato estándar y se define también como único para prevenir conflictos de comunicación y acceso al sistema.

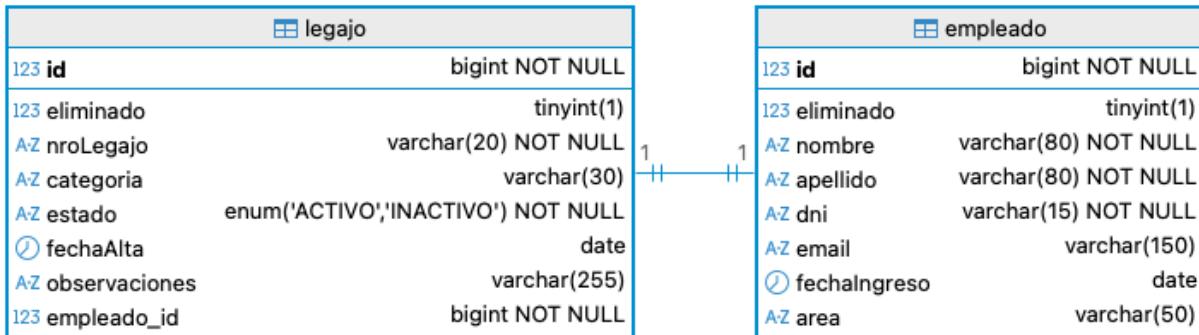
Para reflejar con precisión el ciclo de vida del empleado, el sistema distingue entre dos conceptos clave. Primero, el estado del legajo (ACTIVO o INACTIVO), que define la situación contractual de la persona. Segundo, una bandera de eliminado a nivel de empleado, un mecanismo de borrado lógico reservado para gestionar registros creados por error, ocultándose de la operativa diaria sin perder el dato en caso de que se realice una auditoría. Del mismo modo, se diferencian dos fechas fundamentales: la *fechaIngreso* del empleado, que marca el inicio de su relación contractual y la “*fechaAlta*” del legajo, que corresponde a su creación administrativa en el sistema. Esta separación permite modelar con fidelidad el proceso real, donde puede existir un desfase entre ambas fechas, siendo por lo general la primera que se agrega la de la entidad empleado.

Finalmente, se establece una estricta relación uno a uno entre un empleado y su legajo, asegurando que no haya duplicidades. Esta cardinalidad se implementa de forma robusta a nivel físico mediante una restricción UNIQUE sobre la clave foránea del empleado en la tabla de legajos. Esta medida impide que un mismo empleado pueda ser asociado a más de un legajo, garantizando la unicidad de la relación. La integridad se refuerza con una política de borrado en cascada, la cual subraya que un legajo no tiene entidad propia y depende completamente del ciclo de vida de su empleado asociado, automatizando así la consistencia del sistema y evitando datos huérfanos.

Etapa 1 - Modelado y Definición de Constraints

Consigna: Elaborar el DER y el modelo relacional con PRIMARY KEY, FOREIGN KEY, UNIQUE, CHECK y restricciones de dominio.

Diagrama de Entidad-Relación (DER) en base al script .sql:



Decisiones de diseño y constraints (ver 01_esquema.sql):

```

CREATE DATABASE IF NOT EXISTS empleados;
USE empleados;

DROP TABLE IF EXISTS legajo;
DROP TABLE IF EXISTS empleado;

CREATE TABLE empleado (
    id BIGINT AUTO_INCREMENT PRIMARY KEY,
    eliminado BOOLEAN DEFAULT FALSE,
    nombre VARCHAR(80) NOT NULL,
    apellido VARCHAR(80) NOT NULL,
    dni VARCHAR(15) NOT NULL UNIQUE,
    email VARCHAR(150),
    CONSTRAINT chk_email_formato CHECK (email LIKE
        '%@%.%' OR email IS NULL),
    fechaIngreso DATE,
    area VARCHAR(50)
);

CREATE TABLE legajo (
    id BIGINT AUTO_INCREMENT PRIMARY KEY,
    eliminado BOOLEAN DEFAULT FALSE,
    nroLegajo VARCHAR(20) NOT NULL UNIQUE,
    categoria VARCHAR(30),
    estado ENUM('ACTIVO', 'INACTIVO') NOT NULL,
    fechaAlta DATE,
    observaciones VARCHAR(255),
    empleado_id BIGINT UNIQUE NOT NULL,
    FOREIGN KEY (empleado_id) REFERENCES empleado(id) ON
    DELETE CASCADE
);
    
```

Procedemos a realizar las pruebas de consistencia del diseño, intentando inserciones válidas e inválidas y analizando los mensajes de error.

Inserciones válidas:

Inserción Válida N°1:



```
INSERT INTO empleado (nombre, apellido, dni, email,
fechaIngreso, area)
VALUES ('Carlos', 'Sanchez', '30123456',
'carlos.sanchez@empresa.com', '2022-05-10',
'Tecnología');

-- Asumiendo que el ID del empleado anterior es 1
INSERT INTO legajo (nroLegajo, categoria, estado,
fechaAlta, empleado_id)
VALUES ('L001', 'Desarrollador Semi-Senior', 'ACTIVO',
'2022-05-10', 1);
```

Inserción Válida N°2:



```
INSERT INTO empleado (nombre, apellido, dni, email,
fechaIngreso, area)
VALUES ('Ana', 'Rodriguez', '32987654',
'ana.rodriguez@empresa.com', '2023-01-20',
'Administración');

-- Asumiendo que el ID del empleado anterior es 2
INSERT INTO legajo (nroLegajo, categoria, estado,
fechaAlta, empleado_id)
VALUES ('L002', 'Asistente Administrativa', 'ACTIVO',
'2023-01-20', 2);
```

Captura de pantalla con las inserciones válidas en la tabla empleado:

The screenshot shows the MySQL Workbench interface with the 'empleado' table selected. The table has columns: id, eliminado, nombre, apellido, dni, email, fechalngreso, and area. Two rows are present:

	id	eliminado	nombre	apellido	dni	email	fechalngreso	area
1	1	0	Carlos	Sanchez	30123456	carlos.sanchez@empresa.com	2022-05-10	Tecnología
2	2	0	Ana	Rodriguez	32987654	ana.rodriguez@empresa.com	2023-01-20	Administración

Captura de pantalla con las inserciones válidas en la tabla legajo:

The screenshot shows the MySQL Workbench interface with the 'legajo' table selected. The table has columns: id, eliminado, nroLegajo, categoria, estado, fechaAlta, observaciones, and empleado_id. Two rows are present:

	id	eliminado	nroLegajo	categoria	estado	fechaAlta	observaciones	empleado_id
1	1	0	L001	Desarrollador Semi-Senior	ACTIVO	2022-05-10	[NULL]	1
2	2	0	L002	Asistente Administrativa	ACTIVO	2023-01-20	[NULL]	2

Inserciones inválidas:

Inserción Errónea N°1: Violación de Restricción UNIQUE: Intentamos insertar un nuevo empleado con un DNI que ya existe (30123456). Esto debería activar el error de clave única.



```
INSERT INTO empleado (nombre, apellido, dni, email,
fechaIngreso, area)
VALUES ('Laura', 'Gomez', '30123456',
'laura.gomez@empresa.com', '2024-02-15', 'Ventas');
```

SQL Error [1062] [23000]: Duplicate entry '30123456' for key 'dni'

Inserción Errónea N°2: Violación de Restricción CHECK: Ahora, intentamos insertar un empleado con un formato de email incorrecto (le falta el @). Esto debería activar el error de la restricción CHECK.



```
INSERT INTO empleado (nombre, apellido, dni, email,  
fechaIngreso, area)  
VALUES ('Jorge', 'Martinez', '35111222',  
'jorge.martinez.empresacom', '2024-03-01', 'Soporte');
```

Mensaje de error esperado (MySQL): Check constraint
'chk_email_formato' is violated.

SQL Error [4025] [23000]: CONSTRAINT `chk_email_formato`
failed for `empleados`.`empleado`

Etapa 2 – Generación y carga de datos masivos con SQL puro

Consigna: Generación de datos masivos con SQL puro (entre 10.000 y 500.000 filas distribuidas en 2 tablas).

Descripción conceptual del mecanismo elegido

De acuerdo a nuestras consultas al LLM utilizado sobre el mejor mecanismo posible para generar la cantidad de registros solicitada (nos quedamos con 400000), nuestra elección fue por el mecanismo “Tablas “semilla” y producto cartesiano controlado”. En rigor, el mecanismo fue el de una estrategia híbrida que combina el uso de tablas semilla (catálogos) con funciones nativas de SQL para la generación de datos dinámicos. Consideramos que esta interacción fue una grata experiencia respecto a nuestro proceso de aprendizaje en el uso de SQL y la generación de datos de prueba en futuros proyectos.

El primer paso consistió en la creación de cuatro tablas maestras o "semilla", cuyo propósito fue servir como un catálogo de valores permitidos y realistas. Estas tablas fueron:

- nombres_semilla: Contiene una lista de nombres comunes para asignar a los empleados.
- apellidos_semilla: Contiene una lista de apellidos frecuentes.
- areas_semilla: Define las áreas funcionales válidas dentro de la empresa (ej. 'Tecnología', 'Ventas').
- categorias_semilla: Establece las categorías contractuales posibles para cada legajo (ej. 'Junior', 'Senior').

Estas tablas se poblaron mediante el script 02_catalogos.sql y se utilizaron en el script de carga masiva para seleccionar valores de forma aleatoria (ORDER BY RAND() LIMIT 1), garantizando variedad y coherencia en los datos.

Reglas y Proporciones en la Distribución de Valores

Para que los datos no fueran uniformes y simularán mejor la realidad, se aplicaron las siguientes reglas:

- Asignación de Fechas: La fechalngreso de cada empleado se generó como una fecha aleatoria dentro de un rango de 10 años hacia atrás desde la fecha

actual, utilizando la función CURDATE() - INTERVAL FLOOR(RAND() * 3650) DAY.

- Distribución de Estados de Legajo: Se estableció una proporción para el campo estado en la tabla legajo. Se simuló que la gran mayoría de los empleados estarían activos, asignando el estado 'ACTIVO' a aproximadamente el 90% de los registros y 'INACTIVO' al 10% restante. Esto se logró con la lógica IF(RAND() < 0.9, 'ACTIVO', 'INACTIVO').
- Generación de Datos Únicos: Para campos como dni y email, que requieren unicidad, se combinó un número aleatorio con un identificador secuencial único (ROW_NUMBER()). Esto aseguró que no hubiera colisiones, incluso en un gran volumen de datos.

Garantía de Integridad y Cardinalidades

La consistencia de los datos fue una prioridad. Se garantiza de la siguiente manera:

- Integridad Referencial: La carga se realizó en dos fases secuenciales. Primero, se insertaron los 100,000 registros en la tabla "empleado". Inmediatamente después, se pobló la tabla legajo recorriendo los empleados recién creados y utilizando su empleado.id como clave foránea (FOREIGN KEY). Este método garantiza por diseño que no existan legajos huérfanos, ya que cada legajo se crea a partir de un empleado preexistente.
- Cardinalidad (Relación 1 a 1): La relación uno a uno entre empleado y legajo se aseguró de dos formas:
 - Lógica de Inserción: El script inserta exactamente un registro en legajo por cada registro en "empleado".
 - Restricción de Base de Datos: La columna empleado_id en la tabla legajo tiene una restricción UNIQUE, lo que impide a nivel de base de datos que un mismo empleado pueda tener más de un legajo asociado.
- Restricciones UNIQUE y PRIMARY KEY:
 - Las claves primarias (id) fueron gestionadas automáticamente por el motor de base de datos (AUTO_INCREMENT).
 - La unicidad del dni, email y nroLegajo se aseguró mediante la concatenación de valores aleatorios con la secuencia numérica única generada por ROW_NUMBER(), evitando duplicados.

Scripts que corresponden a la Etapa 2:

Ver **02_catalogos.sql**: Su función es crear y poblar las tablas auxiliares que servirán como fuente de datos para la carga masiva de las dos tablas principales (empleado, legajo).



```
USE empleados;

-- Se eliminan las tablas semilla para poder re-ejecutar el script sin errores
DROP TABLE IF EXISTS nombres_semilla;
DROP TABLE IF EXISTS apellidos_semilla;
DROP TABLE IF EXISTS areas_semilla;
DROP TABLE IF EXISTS categorias_semilla;

-- Creación y carga de la tabla de nombres
CREATE TABLE nombres_semilla (nombre VARCHAR(80));
INSERT INTO nombres_semilla (nombre) VALUES
('Juan'), ('Sofía'), ('Mateo'), ('Valentina'), ('Lucas'), ('Camila'),
('Martín'), ('Isabella'), ('Daniel'), ('Julieta'), ('Agustín'), ('Micaela');

-- Creación y carga de la tabla de apellidos
CREATE TABLE apellidos_semilla (apellido VARCHAR(80));
INSERT INTO apellidos_semilla (apellido) VALUES
('García'), ('Martínez'), ('Rodríguez'), ('López'), ('González'), ('Pérez'),
('Sánchez'), ('Romero'), ('Díaz'), ('Fernández'), ('Gómez'), ('Torres');

-- Creación y carga de la tabla de áreas
CREATE TABLE areas_semilla (area VARCHAR(50));
INSERT INTO areas_semilla (area) VALUES
('Tecnología'), ('Recursos Humanos'), ('Marketing'), ('Ventas'), ('Finanzas');

-- Creación y carga de la tabla de categorías
CREATE TABLE categorias_semilla (categoria VARCHAR(30));
INSERT INTO categorias_semilla (categoria) VALUES
('Junior'), ('Semi-Senior'), ('Senior'), ('Gerente'), ('Analista');
```

Ver **03_carga_masiva.sql**: Su única responsabilidad es usar los datos de los catálogos (creados en el script anterior) para poblar las tablas principales (empleado y legajo) con cientos de miles de registros.



```
USE empleados;

-- POBLANDO LA TABLA `empleado`
INSERT INTO empleado (nombre, apellido, dni, email, fechaIngreso, area)
WITH seq AS (
    -- Generamos una secuencia de números que nos servirá como base para crear datos únicos.
    SELECT (ROW_NUMBER() OVER ()) AS n
    FROM information_schema.columns a, information_schema.columns b
    LIMIT 400000 -- Límite de empleados a generar
)
SELECT
    (SELECT nombre FROM nombres_sevilla ORDER BY RAND() LIMIT 1),
    (SELECT apellido FROM apellidos_sevilla ORDER BY RAND() LIMIT 1),
    CONCAT(CAST(FLOOR(10000000 + RAND() * 90000000) AS CHAR), s.n),
    LOWER(CONCAT(
        (SELECT nombre FROM nombres_sevilla ORDER BY RAND() LIMIT 1), '.',
        (SELECT apellido FROM apellidos_sevilla ORDER BY RAND() LIMIT 1), s.n,
        '@empresa.com'
    )),
    CURDATE() - INTERVAL FLOOR(RAND() * 3650) DAY,
    (SELECT area FROM areas_sevilla ORDER BY RAND() LIMIT 1)
FROM
    seq s;
-- POBLANDO LA TABLA `legajo`
INSERT INTO legajo (nroLegajo, categoria, estado, fechaAlta, observaciones, empleado_id)
SELECT
    CONCAT('LEG-', LPAD(e.id, 6, '0')),
    (SELECT categoria FROM categorias_sevilla ORDER BY RAND() LIMIT 1),
    IF(RAND() < 0.9, 'ACTIVO', 'INACTIVO'),
    e.fechaIngreso,
    'Legajo creado automáticamente.',
    e.id
FROM
    empleado e
WHERE
    e.eliminado = FALSE;
```

Pruebas de consistencia

1. Fragmento .sql para verificación de conteos iniciales:



```
-- Contar el total de empleados
SELECT COUNT(*) AS total_empleados FROM empleado;
-- Contar el total de legajos
SELECT COUNT(*) AS total_legajos FROM legajo;
```

2. Fragmento .sql para verificación de claves Foráneas (FK Huérfanas):



```
-- Buscar legajos cuyo 'empleado_id' no corresponda a ningún empleado existente
SELECT COUNT(*) AS legajos_huerfanos
FROM legajo l
LEFT JOIN empleado e ON l.empleado_id = e.id
WHERE e.id IS NULL;
```

3. Fragmento .sql para verificación de cardinalidad (Uno a Uno):



```
-- Buscar si algún empleado tiene más de un legajo asociado
SELECT empleado_id, COUNT(*)
FROM legajo
GROUP BY empleado_id
HAVING COUNT(*) > 1;
```

4. Fragmento .sql para verificación de distribución de datos:



```
-- Revisar el porcentaje de legajos activos e inactivos
SELECT
    estado,
    COUNT(*) AS cantidad,
    CONCAT(ROUND((COUNT(*) / (SELECT COUNT(*) FROM legajo)) * 100, 2), '%')
AS porcentaje
FROM legajo
GROUP BY estado;
```

5. Fragmento .sql para verificación de unicidad (UNIQUE):



```
-- Buscar si existen DNI duplicados
SELECT dni, COUNT(*)
FROM empleado
GROUP BY dni
HAVING COUNT(*) > 1;
```

Cuadro de verificaciones, teniendo en cuenta los fragmentos de código previos:

Verificación	Objetivo	Consulta Ejecutada	Resultado Obtenido
Conteo Total	Confirmar que se generó la cantidad planificada de registros en las tablas principales.	Fragmento N°1	Empleados: 400000 Legajos: 400000
FK Húrfanas	Asegurar que cada legajo está asociado a un empleado existente (integridad referencial).	Fragmento N°2	El contador devolvió 0. (legajos_huerfanos=0)
Cardinalidad 1-1	Validar que ningún empleado tenga más de un legajo, respetando la relación uno a uno.	Fragmento N°3	La consulta devuelve un conjunto vacío.
Proporción de datos	Verificar que la distribución del campo estado en legajo sigue la regla del 90% 'ACTIVO', lo que sería una proporción generalmente esperable en la realidad y también fue implementado en '03_carga_masiva.sql' .	Fragmento N°4	ACTIVO: 360,247 (90.06%) INACTIVO: 39,753 (9.94%)
Verificación de unicidad	Elegimos utilizar 'DNI' para la prueba de unicidad, que es uno de los campos que utiliza UNIQUE. Aquí buscamos confirmar que no existen DNI duplicados en la tabla empleado.	Fragmento N°5	La consulta devuelve un conjunto vacío.

Etapa 3 – Consultas Avanzadas y Reportes

Consigna: Desarrollo de consultas complejas con JOIN, GROUP BY, HAVING, subconsultas y vistas (Ver 05_consultas.sql y 06_vistas.sql).

Actividades mínimas :

- Diseñar 4 consultas: al menos 2 con JOIN (entre A↔B y/o con tablas auxiliares del dominio), 1 con GROUP BY + HAVING y 1 con subconsulta.

Consultas propuestas:

- Muestra empleados con legajos inactivos.
 - Muestra los empleados junto con su legajo, categoría y estado.
 - Promedio de antigüedad (años) por área.
 - Lista empleados activos (según legajo) que trabajan en un área específica.
 - Área con más personal.
- Crear al menos 1 vista útil en el contexto del sistema.
 - Vista de empleados activos.
 - Vista para reportes generales (sin datos sensibles).
 - Documentar brevemente (3–5 líneas) la utilidad práctica de cada consulta.

Muestra empleados con legajos inactivos:

```
● ● ●  
  
SELECT  
    e.id,  
    e.nombre,  
    e.apellido,  
    e.area,  
    l.nroLegajo,  
    l.estado  
FROM empleado e  
JOIN legajo l ON e.id = l.empleado_id  
WHERE l.estado = 'INACTIVO';
```

Esta consulta permite identificar a los empleados cuyo legajo se encuentra en estado INACTIVO, lo cual puede reflejar bajas temporales, licencias prolongadas o desvinculaciones del personal. Su valor dentro del trabajo radica en el control y auditoría de datos relacionados con la situación laboral de cada empleado. Además, facilita el seguimiento de los registros de legajos que ya no están activos, permitiendo a la organización mantener actualizada la información de su plantilla y aplicar correctamente políticas de reactivación o eliminación lógica de registros.

Muestra los empleados junto con su legajo, categoría y estado



SELECT

```
e.id,  
e.nombre,  
e.apellido,  
e.area,  
l.nroLegajo,  
l.categoría,  
l.estado  
FROM empleado e  
JOIN legajo l ON e.id = l.empleado_id  
ORDER BY e.area, l.categoría;
```

Esta consulta realiza un JOIN entre las tablas empleados y legajo, mostrando en un mismo resultado los datos personales junto con la información administrativa correspondiente. Su importancia reside en ofrecer una visión integral del recurso humano, combinando aspectos personales (nombre, área, fecha de ingreso) con detalles laborales (número de legajo, categoría y estado). Este tipo de consulta es fundamental para generar reportes de gestión, validar la coherencia entre ambas tablas y asegurar la integridad referencial del modelo relacional.

Promedio de antigüedad (años) por área



SELECT

```
e.area,  
ROUND(AVG(TIMESTAMPDIFF(YEAR, e.fechaIngreso, CURDATE())), 2) AS antiguedad_promedio  
FROM empleado e  
GROUP BY e.area  
ORDER BY antiguedad_promedio DESC;
```

El cálculo del promedio de antigüedad por área aporta un análisis estadístico relevante para la toma de decisiones en recursos humanos. Permite evaluar en qué sectores la empresa presenta mayor retención de personal y en cuáles existe una mayor rotación o incorporación reciente. Este tipo de reporte agrega valor al trabajo porque demuestra la capacidad de aplicar funciones de agregación en SQL, al mismo tiempo que genera información práctica para optimizar estrategias de capacitación, promociones o redistribución del personal.

Lista empleados activos (según legajo) que trabajan en un área específica



SELECT

```
e.nombre,  
e.apellido,  
e.area,  
l.nroLegajo,  
l.estado  
FROM empleado e  
JOIN legajo l ON e.id = l.empleado_id  
WHERE l.estado = 'ACTIVO'  
AND e.area = 'Ventas';
```

Esta consulta filtra los empleados en base al estado de su legajo (ACTIVO) y a un área determinada, permitiendo generar reportes personalizados según el departamento o sector que se deseé analizar. Es especialmente útil para controlar la dotación de personal vigente dentro de un área, verificar que las asignaciones estén actualizadas y facilitar el seguimiento operativo. En el contexto del trabajo, demuestra el manejo de consultas combinadas con JOIN y condiciones lógicas aplicadas a dos tablas relacionadas.

Área con más personal



```
SELECT e.area, COUNT(*) AS cantidad_empleados
FROM empleado e
GROUP BY e.area
HAVING COUNT(*) = (
    SELECT MAX(cantidad)
    FROM (
        SELECT COUNT(*) AS cantidad
        FROM empleado
        GROUP BY area
    ) AS sub_conteo_por_area
);
```

Esta consulta, que combina agrupamiento y una subconsulta o cláusula ORDER BY COUNT(), permite identificar cuál es el área que concentra la mayor cantidad de empleados. Su relevancia en el trabajo radica en mostrar cómo el uso de funciones de agregación y análisis comparativo puede generar información estratégica. Saber qué área tiene más personal es clave para la planificación de recursos, la evaluación de carga laboral y el análisis de estructura organizacional.

Vista de empleados activos



```
CREATE OR REPLACE VIEW vista_empleados_activos AS
SELECT
    e.id AS id_empleado,
    e.nombre,
    e.apellido,
    e.area,
    e.fechaIngreso,
    l.nroLegajo,
    l.categoría,
    l.estado
FROM empleado e
JOIN legajo l ON e.id = l.empleado_id
WHERE l.estado = 'ACTIVO';
```

La vista denominada “vista_empleados_activos” fue creada con el objetivo de simplificar las consultas frecuentes sobre el personal que se encuentra actualmente en actividad dentro de la organización. Al unir las tablas empleados y legajo, y filtrar únicamente los registros cuyo estado es ‘ACTIVO’, esta vista ofrece una forma rápida y eficiente de acceder a información relevante sin necesidad de escribir consultas complejas cada vez. Su principal valor radica en facilitar la generación de reportes de recursos humanos, control de dotación vigente y análisis operativo del personal. Además, permite mantener una visión actualizada del estado del personal sin afectar las tablas originales, contribuyendo a una gestión más ordenada y segura de los datos.

Vista para reportes generales (sin datos sensibles)



```
CREATE OR REPLACE VIEW vista_empleados_publica AS
SELECT
    e.nombre,
    e.apellido,
    e.area,
    l.categoría,
    l.estado
FROM empleado e
JOIN legajo l ON e.id = l.empleado_id
```

La vista “vista_empleados_publica” se diseñó con el propósito de proteger la información sensible del personal (como DNI, correo electrónico o datos internos), al mismo tiempo que proporciona los campos necesarios para elaborar reportes generales de gestión. De esta manera, se pueden generar listados o estadísticas sin comprometer la privacidad ni la seguridad de los empleados. Esta vista aporta un valor agregado al trabajo práctico al demostrar la aplicación de buenas prácticas en seguridad de bases de datos, mediante la restricción del acceso a datos confidenciales y la exposición controlada de la información relevante. Además, facilita la colaboración entre distintas áreas del sistema, que pueden consultar los datos sin riesgos de alterar información crítica.

- Medición comparativa con/sin índice en 3 consultas (igualdad, rango, JOIN) con breve conclusión (Ver 04_indices.sql y 05_explain.sql)



```
-- -----  
-- Archivo: 04_indices.sql  
-- TFI - Bases de Datos I  
  
-- Propósito: Crea los índices utilizados en la Etapa 3  
para  
--           la medición comparativa de rendimiento de  
consultas.  
-- -----  
-- -----  
  
USE empleados;  
  
-- Índice para la consulta por igualdad en 'dni'  
CREATE INDEX idx_empleado_dni ON empleado(dni);  
  
-- Índice para la consulta por rango en 'fechaIngreso'  
CREATE INDEX idx_empleado_fechaIngreso ON  
empleado(fechaIngreso);  
  
-- Índice para la consulta con JOIN en la clave foránea  
'empleado_id'  
CREATE INDEX idx_legajo_empleado_id ON  
legajo(empleado_id);  
  
-- Índice adicional para optimizar la búsqueda por estado  
CREATE INDEX idx_legajo_estado ON legajo(estado);
```



```
-- 1. Consulta por igualdad (dni)
-- Sin índice:
EXPLAIN SELECT * FROM empleado WHERE dni = '30555666';
-- 2. Consulta por rango (fechaIngreso)
-- Sin índice:
EXPLAIN SELECT * FROM empleado
WHERE fechaIngreso BETWEEN '2015-01-01' AND '2020-12-31';
-- 3. Consulta con JOIN (estado y empleado_id)
-- Sin índice:
EXPLAIN
SELECT e.nombre, e.apellido, l.categoría, l.estado
FROM empleado e
JOIN legajo l ON e.id = l.empleado_id
WHERE l.estado = 'ACTIVO';
```



```
-- Con índice (idx_empleado_dni):
EXPLAIN SELECT * FROM empleado WHERE dni = '30555666';
-- Con índice (idx_empleado_fechaIngreso):
EXPLAIN SELECT * FROM empleado
WHERE fechaIngreso BETWEEN '2023-01-01' AND '2023-01-31';
-- Con índice (idx_legajo_empleado_id):
EXPLAIN
SELECT e.nombre, e.apellido, l.categoría, l.estado
FROM empleado e
JOIN legajo l ON e.id = l.empleado_id
WHERE l.estado = 'INACTIVO';
```

En esta sección se realizó una medición comparativa del rendimiento de consultas SQL con y sin el uso de índices, con el objetivo de analizar el impacto que tienen sobre la eficiencia en la búsqueda y recuperación de información. Los índices son estructuras adicionales que permiten al motor de base de datos acceder más rápidamente a los registros, evitando recorridos completos de las tablas. Para comprobar su efecto, se seleccionaron tres tipos de consultas representativas: una de igualdad sobre el campo dni, una de rango sobre “fechalngreso”, y una consulta con JOIN entre las tablas empleados y legajo. En cada caso se ejecutó primero la

consulta sin índices, y luego se repitió tras la creación del índice correspondiente, utilizando el comando EXPLAIN para observar los planes de ejecución y la cantidad de filas analizadas. Esta comparación permitió evaluar de forma empírica cómo la implementación de índices contribuye a mejorar el rendimiento y la optimización del sistema de base de datos.

En las siguientes pruebas se ejecutarán tres consultas distintas con el fin de comparar su rendimiento antes y después de aplicar índices. La primera prueba consiste en una consulta por igualdad, donde se busca un empleado específico utilizando el campo dni. La segunda corresponde a una consulta por rango, que filtra empleados según su fecha de ingreso dentro de un intervalo determinado. Finalmente, la tercera evalúa una consulta con JOIN, en la cual se combinan las tablas empleados y legajo para obtener información conjunta. En cada caso, se observará cómo la creación de índices optimiza el acceso a los datos y mejora el tiempo de respuesta del sistema.

A partir de las mediciones realizadas, se pudo comprobar que la incorporación de índices produce una mejora significativa en la velocidad de ejecución y eficiencia de las consultas. En las búsquedas por igualdad (dni), el motor pasó de realizar un escaneo completo de la tabla (type = ALL) a usar el índice (type = ref), reduciendo drásticamente el número de filas leídas. En las consultas por rango (fechalngreso), el índice permitió que la base de datos aplicará una búsqueda selectiva, optimizando la recuperación de registros dentro de intervalos específicos. Finalmente, en las consultas con JOIN, el índice sobre la clave foránea (empleado_id) mejoró la forma en que se realizan las uniones entre tablas, evitando comparaciones innecesarias.

En conclusión, los resultados evidencian que el uso adecuado de índices no solo reduce los tiempos de respuesta, sino que también mejora la escalabilidad y rendimiento general del sistema, resultando una práctica esencial en el diseño y mantenimiento de bases de datos relacionales.

Análisis de los resultados, tabla resumen en consultas sin índices:

Tipo de Consulta	Tabla	Tipo de Acceso (type)	Filas Estimadas (rows)	Observaciones
Igualdad (dni)	empleado	const	1	Fila no encontrada
Rango (fechalngreso)	empleado	ALL	397,475	Escaneo completo de la tabla.
JOIN (estado)	legajo (l)	ALL	397,621	Escaneo completo de la tabla.
	empleado (e)	eq_ref	1	Acceso eficiente por Clave Primaria.

Análisis de los resultados, tabla resumen en consultas con índices:

Tipo de Consulta	Tabla	Tipo de Acceso (type)	Filas Estimadas (rows)	Índice Utilizado (key)	Observaciones
Igualdad (dni)	empleado	const	1	idx_empleado_dni	Éxito: Acceso instantáneo usando el índice.
Range (fechalngreso)	empleado	range	3486	idx_empleado_fechalngreso	Éxito: Se evita el escaneo completo, mejora >99%.
JOIN (estado)	legajo (l)	ref	77,146	idx_legajo_estado	Éxito: Se usa el índice en estado para el filtro.
	empleado (e)	eq_ref	1	PRIMARY	Acceso eficiente por Clave Primaria.

Un ejemplo de cómo poder mejorar el código con la asistencia de la IA

Se realiza una consulta compleja que sería “Áreas ordenadas por la cantidad de empleados activos e inactivos”. Donde buscaremos obtener una respuesta de todas las áreas con su totalidad de empleados, cantidad de empleados activos, empleados inactivos y realizar una operación dentro de la consulta que nos diga qué porcentaje de empleados activos tiene cada área.

El resultado final:

```
● ● ●
SELECT
    e.area,
    COUNT(*) AS total_empleados,
    SUM(CASE WHEN l.estado = 'ACTIVO' THEN 1 ELSE 0 END) AS
    empleados_activos,
    SUM(CASE WHEN l.estado = 'INACTIVO' THEN 1 ELSE 0 END) AS
    empleados_inactivos,
    ROUND(
        SUM(CASE WHEN l.estado = 'ACTIVO' THEN 1 ELSE 0 END) /
        NULLIF(COUNT(*), 0) * 100, 2
    ) AS porcentaje_activos
FROM empleado e
JOIN legajo l ON e.id = l.empleado_id
GROUP BY e.area
ORDER BY empleados_activos DESC, empleados_inactivos ASC;
```

Durante el proceso de optimización, se comenzó con una consulta básica sin agregaciones, que resultaba poco eficiente para obtener información resumida. Luego se incorporó la cláusula GROUP BY para agrupar resultados, aunque inicialmente devolvía filas separadas por estado. En una tercera versión se añadieron expresiones condicionales con CASE WHEN, permitiendo consolidar los datos en una única fila por área. Finalmente, la versión optimizada incorporó el cálculo de totales, porcentajes y un ordenamiento múltiple, logrando una consulta completa, legible y con mejor rendimiento gracias al uso eficiente de funciones de agregación y condiciones dentro del mismo bloque SQL.

Etapa 4 – Seguridad e Integridad

Consigna: Creación de usuarios con privilegios mínimos, vistas para ocultar información sensible, validación de restricciones y consultas seguras en Java con PreparedStatement o equivalente en SQL (ver 07_seguridad.sql).

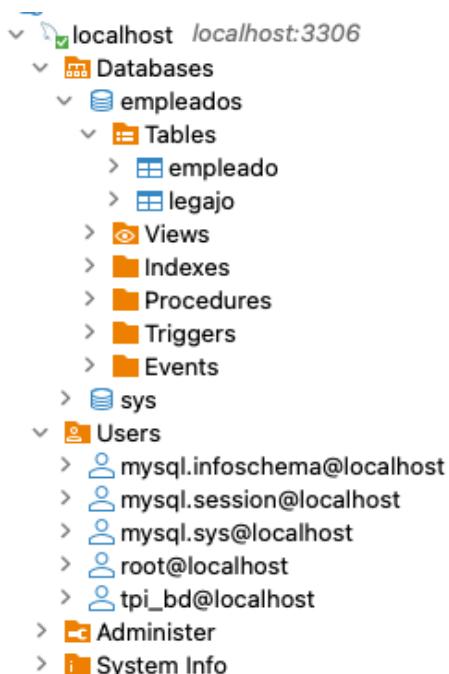
Creación de usuario con privilegios mínimos

Tarea: Crear un usuario con privilegios mínimos y mostrar pruebas de acceso restringido.

Comenzando con la parte de creación de usuario con privilegios mínimos, lo que hacemos es crear un usuario llamado “tpi_bd” para “localhost” (es decir, para el entorno local de quién lo corra en cada caso) con contraseña “1234”:

```
● ● ●  
  
-- 1. Crear el usuario 'tpi_bd' que solo se conecta  
localmente  
CREATE USER IF NOT EXISTS 'tpi_bd'@'localhost' IDENTIFIED  
BY '1234';
```

Captura con el usuario creado luego de ejecutar la línea anterior:



Posteriormente, le otorgamos privilegios mínimos:

```
-- Dar permiso básico para acceder/usar la base de datos
'empleados'
GRANT USAGE ON empleados.* TO 'tpi_bd'@'localhost';

-- 2. Dar permisos CRUD (sin DELETE) sobre las tablas
principales
GRANT SELECT, INSERT, UPDATE ON empleados.empleado TO
'tpi_bd'@'localhost';
GRANT SELECT, INSERT, UPDATE ON empleados.legajo TO
'tpi_bd'@'localhost';
```

Captura de los privilegios en el nuevo usuario

Tabla empleado:

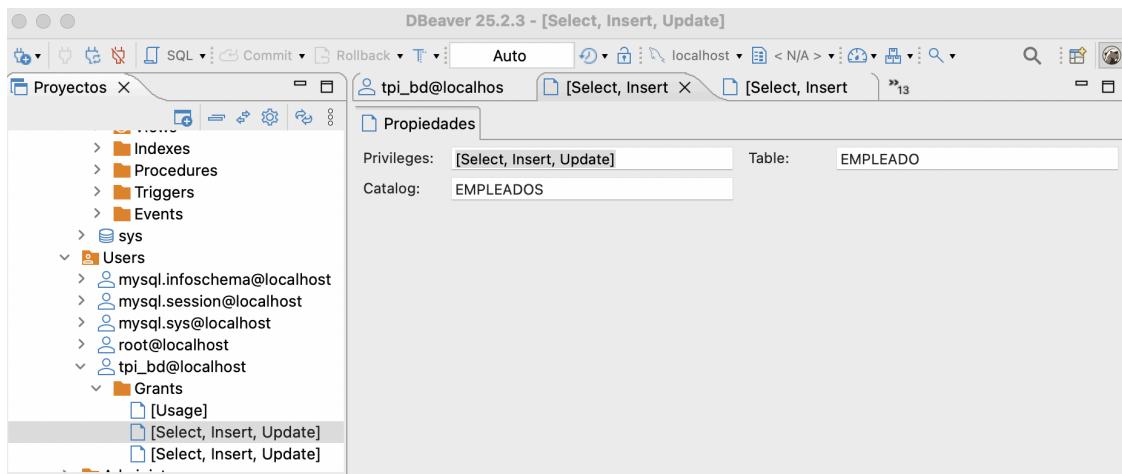
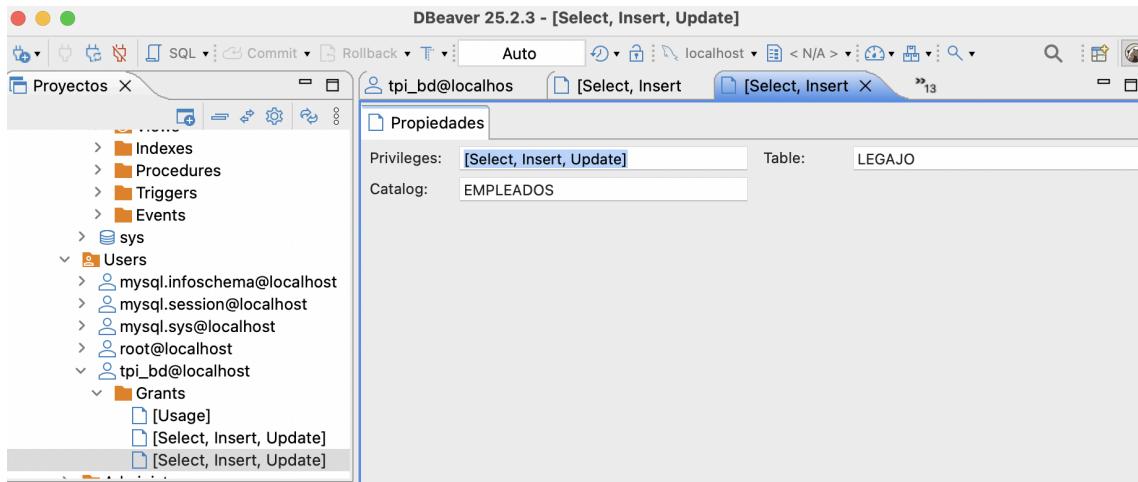


Tabla legajo:



Estos privilegios mínimos comprenden la posibilidad de utilizar SELECT, INSERT Y UPDATE solo en las tablas empleado y legajo. Por lo tanto, ese será el parámetro para realizar las pruebas de acceso restringido.

Antes de realizar estas pruebas, primero se debe ingresar con el usuario creado, ya que por defecto estaríamos usando root que tiene todos los privilegios. Para esto, se puede salir de la conexión actual y crear una nueva visualmente tanto en SQL Workbench como en Dbeaver. En el caso de Dbeaver, fue necesario cambiarlo en Editar Conexión - Driver Properties la opción allowPublicKeyRetrieval de “false” a “true” para poder finalmente rotar de usuario. También es posible realizarlo desde la consola con mysql, saliendo primero de root con “exit” y luego ingresando con el nuevo usuario mediante “mysql -u tpi_bd -p”.

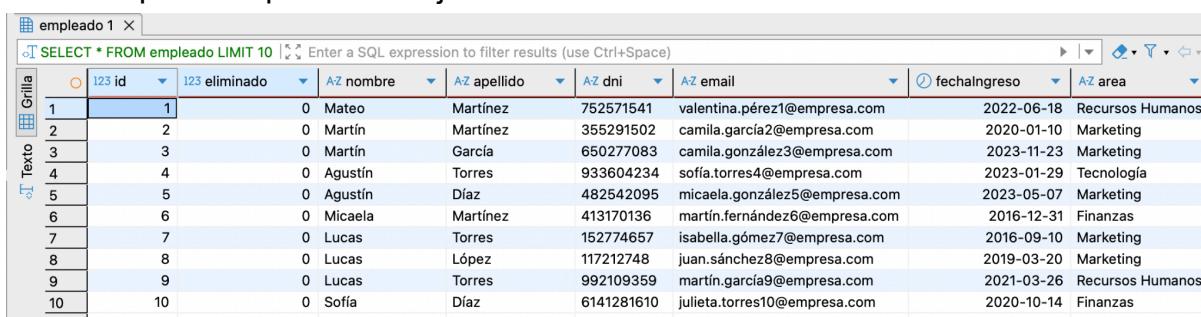
Dicho esto, pasemos a las pruebas:

Prueba N°1 de acceso restringido correcto, a través de la cuál buscamos leer algunos datos de empleado:



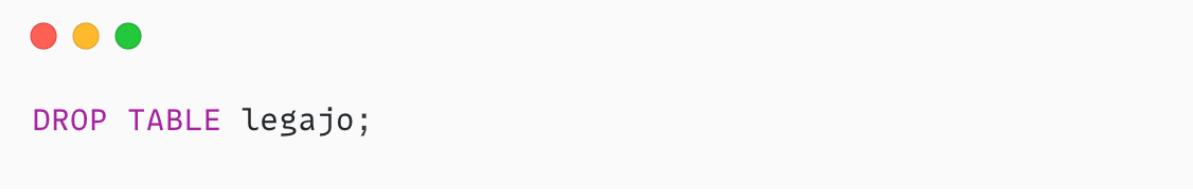
```
SELECT * FROM empleado LIMIT 10;
```

Captura de pantalla al ejecutar la consulta anterior con el nuevo usuario:



	123 id	123 eliminado	AZ nombre	AZ apellido	AZ dni	AZ email	fechalngreso	AZ area
1	1	0	Mateo	Martínez	752571541	valentina.pérez1@empresa.com	2022-06-18	Recursos Humanos
2	2	0	Martín	Martínez	355291502	camila.garcía2@empresa.com	2020-01-10	Marketing
3	3	0	Martín	García	650277083	camila.gonzález3@empresa.com	2023-11-23	Marketing
4	4	0	Agustín	Torres	933604234	sofia.torres4@empresa.com	2023-01-29	Tecnología
5	5	0	Agustín	Díaz	482542095	micaela.gonzález5@empresa.com	2023-05-07	Marketing
6	6	0	Micaela	Martínez	413170136	martín.fernández6@empresa.com	2016-12-31	Finanzas
7	7	0	Lucas	Torres	152774657	isabella gómez7@empresa.com	2016-09-10	Marketing
8	8	0	Lucas	López	117212748	juan.sánchez8@empresa.com	2019-03-20	Marketing
9	9	0	Lucas	Torres	992109359	martín.garcía9@empresa.com	2021-03-26	Recursos Humanos
10	10	0	Sofía	Díaz	6141281610	julieta.torres10@empresa.com	2020-10-14	Finanzas

Prueba N°2, que debe fallar, ya que buscamos hacer un DROP, para lo cual el usuario creado no tiene privilegios:



```
DROP TABLE legajo;
```



```
SQL Error [1142] [42000]: DROP command denied to user
'tpi_bd'@'localhost' for table 'legajo'
```

Captura de pantalla del error al ejecutar DROP con el nuevo usuario:

```
Estadísticas 1 ×
SQL Error [1142] [42000]: DROP command denied to user 'tpi_bd'@'localhost' for table 'legajo'
DROP TABLE legajo
```

Prueba N°3, que debe fallar, ya que el usuario “tpi_bd” no tiene privilegios para leer datos fuera de la base de datos empleados, especialmente en bases de datos sensibles del sistema como mysql:



```
SELECT * FROM mysql.user;
```

Captura del Error (Prueba N°3): Falla esperada al ejecutar SELECT en mysql.user, confirmando la restricción de acceso a bases de datos del sistema.

```
Estadísticas 1 ×
SQL Error [1142] [42000]: SELECT command denied to user 'tpi_bd'@'localhost'
for table 'user'
SELECT * FROM mysql.user
LIMIT 0, 200
```

Diseño de vistas para ocultar información sensible

Tarea: Diseñar 2 vistas que oculten información sensible

Volvemos al usuario root para crear las vistas propuestas. En el primer caso no fue necesario crear una vista nueva, ya que previamente creamos “vista_empleados_publica” que ocultaba algunos datos sensibles. Lo que si le agregamos, en base a nuestra conversación paso a paso con IA, fue “eliminado = false” para que no muestre los registros que estén en esa categoría:



```
USE empleados;

-- VISTA 1: Empleados públicos (actualizada para no mostrar
eliminados)
-- Oculta: id, dni, email, fechaIngreso, observaciones,
empleado_id
CREATE OR REPLACE VIEW vista_empleados_publica AS
SELECT
    e.nombre,
    e.apellido,
    e.area,
    l.categoría,
    l.estado
FROM empleado AS e
JOIN legajo AS l ON e.id = l.empleado_id
WHERE e.eliminado = FALSE AND l.eliminado = FALSE;
```

La segunda vista que oculta información sensible es la nueva “vista_legajos_activos”, que oculta “observaciones” como dato al que interpretamos como confidencial:



```
-- VISTA 2: Legajos Activos (para gerencia, sin
observaciones)
-- Oculta: id, eliminado, observaciones, empleado_id
CREATE OR REPLACE VIEW vista_legajos_activos AS
SELECT
    e.nombre,
    e.apellido,
    l.nroLegajo,
    l.categoría,
    l.estado,
    l.fechaAlta
FROM legajo AS l
JOIN empleado AS e ON l.empleado_id = e.id
WHERE l.estado = 'ACTIVO' -- Solo activos
    AND l.eliminado = FALSE
    AND e.eliminado = FALSE;
```

Por último, le otorgamos permisos al usuario creado para que pueda acceder a estas vistas:



```
-- Otorgar permiso de lectura sobre las vistas al usuario
tpi_bd
GRANT SELECT ON empleados.vista_empleados_publica TO
'tpi_bd'@'localhost';
GRANT SELECT ON empleados.vista_legajos_activos TO
'tpi_bd'@'localhost';
```

De todas formas, vale aclarar que somos conscientes de que el usuario “tpi_bd” mantiene sus privilegios SELECT por sobre la tabla empleados, por lo que puede acceder a las columnas que estas vistas no estarían mostrando y obviando su razón de ser. Estrictamente hablando la consigna no refiere a crear estas dos vistas restringidas para “ocultarle” a este usuario ciertos datos, por lo que preferimos que el usuario mantenga esos privilegios en caso de que sea necesario para las actividades posteriores.

Captura de pantalla de vista_empleados_publica desde el usuario creado:

vista_empleados_publica 1 ×					
SELECT * FROM vista_empleados_r Enter a SQL expression to filter results (use Ctrl+Space)					
	AZ nombre	AZ apellido	AZ area	AZ categoria	AZ estado
1	Mateo	Martínez	Recursos Humanos	Senior	ACTIVO
2	Martín	Martínez	Marketing	Gerente	ACTIVO
3	Martín	García	Marketing	Gerente	ACTIVO
4	Agustín	Torres	Tecnología	Senior	ACTIVO
5	Agustín	Díaz	Marketing	Gerente	ACTIVO

Captura de pantalla de vista_legajos_activos desde el usuario creado:

vista_legajos_activos 1 ×						
SELECT * FROM vista_legajos_activos Enter a SQL expression to filter results (use Ctrl+Space)						
	AZ nombre	AZ apellido	AZ nroLegajo	AZ categoria	AZ estado	fechaAlta
1	Mateo	Martínez	LEG-000001	Senior	ACTIVO	2022-06-18
2	Martín	Martínez	LEG-000002	Gerente	ACTIVO	2020-01-10
3	Martín	García	LEG-000003	Gerente	ACTIVO	2023-11-23
4	Agustín	Torres	LEG-000004	Senior	ACTIVO	2023-01-29
5	Agustín	Díaz	LEG-000005	Gerente	ACTIVO	2023-05-07

Pruebas de integridad

Tarea: Ejecutar al menos 2 pruebas de integridad (ej. duplicación de PK, inserción fuera de rango, violación de FK)

Ejecutamos pruebas de integridad en función de algunas de las reglas que establecimos en la etapa 1, siendo que justamente las pruebas de integridad son para intentar generar errores en función de esas reglas preexistentes en la estructura de la base de datos.

Prueba Nº1: Violación de UNIQUE (Duplicación)

La prueba Nº1 serán en función de la violación del UNIQUE en la columna “dni” de la tabla empleado, intentando colocar dos empleados con un mismo DNI cuando esto no podría ser posible. Para esto, seleccionamos un registro preexistente con un valor en la columna dni de “752571541” y luego ejecutamos INSERT INTO para agregar otro empleado con un mismo DNI:



```
INSERT INTO empleado (nombre, apellido, dni, email,
fechaIngreso, area)
VALUES
('Intento', 'Duplicado', '752571541', 'duplicado@test.com',
CURDATE(), 'Prueba');
```

Captura con el error de duplicación arrojado:

The screenshot shows a MySQL Workbench interface with a statistics window titled "Estadísticas 1". It displays an error message: "SQL Error [1062] [23000]: Duplicate entry '752571541' for key 'empleado.dni'". To the right, the SQL query is shown in the editor:

```
INSERT INTO empleado (nombre, apellido, dni, email, fechaIngreso, area)
VALUES
('Intento', 'Duplicado', '752571541', 'duplicado@test.com', CURDATE(), 'Prueba');
```

Prueba 2: Violación de FOREIGN KEY (FK)

Esta segunda prueba hace referencia al hecho de que “legajo” tiene una FOREIGN KEY en “empleado_id” que apunta a “empleado(id)”. Esto significa que no podemos crear un legajo para un “empleado_id” que no exista en la tabla empleado, algo que también se mencionó previamente en la etapa 1. El procedimiento es similar al de la prueba Nº1, simplemente que cambiamos los valores en INSERT INTO y VALUES con los que nos importan acá, ya que hacemos referencia a la tabla “legajos”:



```
INSERT INTO legajo (nroLegajo, categoria, estado,
fechaAlta, observaciones, empleado_id)
VALUES
('LEG-INVALIDO', 'Prueba FK', 'ACTIVO', CURDATE(), 'Este
empleado no existe', 999999);
```

En la ejecución nos arroja el error esperado:

The screenshot shows the MySQL Workbench interface. On the left, there's a statistics panel titled 'Estadísticas 1' with an error message: 'SQL Error [1452] [23000]: Cannot add or update a child row: a foreign key constraint fails ('empleados'.`legajo`, CONSTRAINT `legajo_ibfk_1` FOREIGN KEY (`empleado_id`) REFERENCES `empleado` (`id`) ON DELETE CASCADE)'. On the right, the SQL editor window contains the same INSERT query. Below the editor, the status bar shows 'Inserción inteligente' and the time '15 : 49 : 270'.

Implementación de consulta parametrizada en Java

Tarea: Implementar en Java al menos 1 consulta parametrizada con PreparedStatement para prevenir SQL Injection o SQL (procedimiento almacenado sin SQL dinámico), con prueba anti-inyección documentada

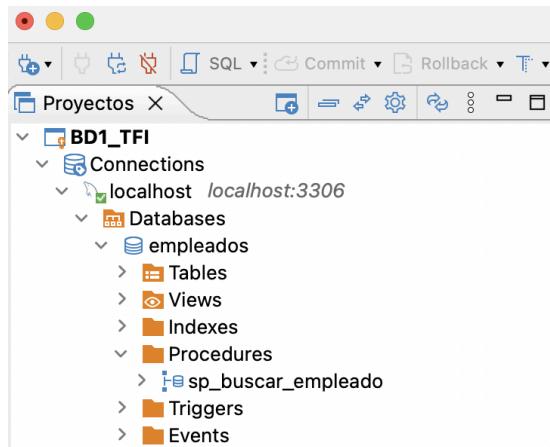
Cómo método para esta tarea, dado que todavía no comenzamos a implementar Java en nuestro proyecto, elegimos solucionarlo con un procedimiento almacenado en SQL.

Para evitar la SQL Injection, creamos este procedimiento almacenado, que es una plantilla de consulta segura y pre-compilada. Este procedimiento obliga a MySQL a tratar la entrada del DNI siempre como un simple texto (datos), nunca como un comando. Así, si un atacante envía ' OR '1'='1', el sistema buscará ese texto literal en lugar de ejecutar el comando, neutralizando el ataque:



```
CREATE PROCEDURE sp_buscar_empleado(IN dni_param VARCHAR(15))
BEGIN
    -- La consulta usa el parámetro de entrada de forma segura
    SELECT * FROM empleado
    WHERE dni = dni_param
    AND eliminado = FALSE; -- Buena práctica! no mostrar eliminados
END;
```

Al ejecutar las líneas previas, se crea el procedimiento en la base de datos:



Ya teniendo el procedimiento creado, procedemos con las pruebas anti-inyección:

```
-- Prueba "de control"; consulta normal que busca el DNI mencionado (ya existente en la base de datos) a través del procedimiento
CALL sp_buscar_empleado('752571541');
```

Captura de pantalla “prueba de control DNI existente”:

empleado 1										Estadísticas 1	
CALL sp_buscar_empleado('752571541')										Enter a SQL expression to filter results (use Ctrl+Space)	
Grilla	123 id	123 eliminado	AZ nombre	AZ apellido	AZ dni	AZ email	fechalngreso	AZ area		Buttons	
1	1	0	Mateo	Martínez	752571541	valentina.pérez1@empresa.com	2022-06-18	Recursos Humanos			
1 row(s) fetched - 0.0s (0.0s fetch), on 2025-10-22 at 13:00:47											

```
-- Prueba Anti-Inyección ; si esta no funciona, el procedimiento funciona correctamente frente a un ataque SQL Injection
```

```
CALL sp_buscar_empleado(" OR '1'='1");
```

Captura de pantalla Anti-Inyección:

The screenshot shows a MySQL Workbench interface with a query editor and a results grid. The query entered is:

```
CALL sp_buscar_empleado(' OR '1'='1')
```

The results grid is empty, indicating no data found. The status bar at the bottom right shows "No data- 0.0s, on 2025-10-22 at 13:02:17".

Justamente la tabla nos aparece vacía, ya que es lo que tiene que suceder. Mientras que la prueba de control nos arrojó el registro buscado mediante en dni ingresado, la prueba anti-inyección nos señala que un atacante no puede buscar ese registro mediante el ingreso de '`OR '1'='1`', que es el método por el cual el SQL Injection funciona para poder usar WHERE indebidamente de parte del atacante.

Etapa 5 – Concurrency y Transacciones

Simulación de bloqueos y deadlocks; niveles de aislamientos; transacciones en Java o SQL (procedimientos almacenados) con retry ante deadlock.

- Simular al menos 1 deadlock en dos sesiones y documentar el error generado. (ver [09_concurrencia_guiada.sql](#)).

Simulación de Deadlock en dos sesiones

El objetivo de esta prueba es provocar deliberadamente un interbloqueo (deadlock) para observar y documentar cómo el motor de la base de datos (MySQL) maneja esta situación de bloqueo circular. Un deadlock ocurre cuando dos o más transacciones se bloquean mutuamente, cada una esperando un recurso que la otra posee.

Para realizar esta tarea, fue necesaria una ejecución paso a paso, intercalada en dos scripts distintos que simularon las dos sesiones, con la finalidad de crear la dependencia circular necesaria para generar el error.

En nuestro caso, que en general usamos Dbeaver, hubo que desactivar el modo Auto-Commit para ambos scripts, pasándolo a "Commit Manual", ya que sino no era posible efectuar esta tarea. Para esto los pasos a seguir en DBeaver son, estando en la pestaña de cada uno de los scripts: Base de Datos - Modo de transacción - Commit Manual (Repeatable Read)

```
● ● ●

-- Sesión 1
USE empleados;
START TRANSACTION;
UPDATE empleado SET area = 'Ventas' WHERE id = 1234;
UPDATE legajo SET observaciones = 'Se transfirió al área de Ventas'
WHERE empleado_id = 1234;
```

```
SQL terminal:  
> USE empleados  
0 row(s) modified.  
> START TRANSACTION  
0 row(s) modified.  
> UPDATE empleado SET area = 'Ventas' WHERE id = 1234  
1 row(s) modified.  
> UPDATE legajo SET observaciones = 'Se transfirió al área de  
Ventas' WHERE empleado_id = 1234  
  
SQL Error [1213] [40001]: Deadlock found when trying to get lock;  
try restarting transaction
```

```
-- Sesión 2  
USE empleados;  
START TRANSACTION;  
UPDATE legajo SET observaciones = 'Se requiere actualizar sueldo'  
WHERE empleado_id = 1234;  
UPDATE empleado SET area = 'Ventas' WHERE id = 1234;
```

```
SQL terminal  
> USE empleados  
0 row(s) modified.  
> START TRANSACTION  
0 row(s) modified.  
> UPDATE legajo SET observaciones = 'Se requiere actualizar sueldo'  
WHERE empleado_id = 1234  
1 row(s) modified.  
> UPDATE empleado SET area = 'Ventas' WHERE id = 1234  
1 row(s) modified.
```

Captura de pantalla luego del error de Deadlock:



La sesión 1 primero bloquea al empleado (UPDATE empleado) y la sesión 2 bloquea al legajo (UPDATE legajo) del mismo id.

Cuando la sesión 1 intenta bloquear a continuación el legajo, se queda esperando, no finaliza la acción, lo que puede generar el error de tiempo de espera excedido.

El error de Deadlock aparece porque cuando se ejecuta la última línea del script de la sesión 2 intentando actualizar al empleado, en la sesión 1 arroja el error de Deadlock.

MySQL detecta este ciclo imposible y sacrifica una de las sesiones arrojando el error 1213 que en nuestro caso fue la sesión 1.

Las filas permanecen bloqueadas si al final no se realiza un COMMIT (confirma los cambios) o un ROLLBACK (revierte los cambios).

- Implementar en Java o en SQL (procedimiento almacenado) un ejemplo de transacciones con START TRANSACTION/COMMIT/ROLLBACK y logging de errores. Si hay deadlock (1213/SQLSTATE 40001), aplicar retry hasta 2 intentos con backoff breve. (ver 08_transacciones.sql).



```

USE empleados;

-- Tabla de registro de errores
CREATE TABLE IF NOT EXISTS registro_errores (
    id INT AUTO_INCREMENT PRIMARY KEY,
    fecha TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    mensaje VARCHAR(200)
);

DELIMITER $$

CREATE PROCEDURE transferir_area(
    IN transferir_empleado_id INT,
    IN nueva_area VARCHAR(30),
    IN observacion VARCHAR(200)
)
BEGIN
    DECLARE intentos INT DEFAULT 0;
    DECLARE max_intentos INT DEFAULT 2;
    DECLARE continuar INT DEFAULT 1;

    -- Handler para DEADLOCK (1213)
    DECLARE CONTINUE HANDLER FOR 1213
    BEGIN
        ROLLBACK;
        SET autocommit = 1;
        INSERT INTO registro_errores (mensaje)
        VALUES ('Error de Deadlock (1213)');
        SET autocommit = 0;
        SET intentos = intentos + 1;
        DO SLEEP(1 + RAND());
    END;

    -- Handler para cualquier otro error
    DECLARE EXIT HANDLER FOR SQLEXCEPTION
    BEGIN
        ROLLBACK;
        SET autocommit = 1;
        INSERT INTO registro_errores (mensaje)
        VALUES ('Error General');
        SET autocommit = 0;
        SET continuar = 0;
    END;

    -- Bucle de reintentos
    bucle_intentos: WHILE intentos < max_intentos AND continuar =
1 DO
    START TRANSACTION;

    UPDATE empleado
    SET area = nueva_area
    WHERE id = transferir_empleado_id;

    UPDATE legajo
    SET observaciones = CONCAT(IFNULL(observaciones, ''), ' | '
    , observacion)
    WHERE empleado_id = transferir_empleado_id;

    COMMIT;

    SET continuar = 0;
END WHILE bucle_intentos;

END$$

DELIMITER ;

```

El siguiente paso a ejecutar el script anterior (08_transacciones.sql) es verificar que funciona correctamente.

Una vez creado el procedimiento, lo llamamos con los parámetros indicados:



```
CALL transferir_area(1234, 'Tecnología', 'El empleado se transfiere de área, actualizar salario');
```

Para generar un error de Deadlock, se necesita otra vez abrir dos sesiones para repetir el procedimiento anterior, bloqueando las filas utilizadas a través del procedimiento (ver 09_concurrencia_guiada.sql).



```
-- Sesión 1
START TRANSACTION;
UPDATE legajo SET observaciones = 'Esto es un intento de Error de Deadlock' WHERE empleado_id = 3;

-- Sesión 2
CALL transferir_area(3, 'Marketing', 'Transferencia a Marketing');

-- Sesión 1
UPDATE empleado SET area = 'Recursos Humanos' WHERE id = 3;
COMMIT;
```

De esta forma se provoca el error y queda registrado conforme el procedimiento. Para consultar el registro de errores usamos SELECT * FROM registro_errores; y podemos observar la tabla de la siguiente manera.

	Grilla	123 ↗ id	fecha	A-Z mensaje
1	1	2025-10-21 17:43:25	Error de Deadlock (1213)	

Para generar un error general usamos el siguiente código. En este caso, forzamos un error que no es un deadlock para demostrar que el handler de errores genérico funciona. Aquí no encontrará la tabla legajo, ya que su nombre se cambia temporalmente (ver en 08_transacciones.sql):


```
RENAME TABLE legajo TO legajo_temp;
CALL transferir_area(1234, 'Finanzas', 'Cambio de prueba');
RENAME TABLE legajo_temp TO legajo;
```

Para ver el listado de errores generados en orden descendente por fecha usamos:



```
SELECT * FROM registro_errores ORDER BY fecha DESC;
```

	Grilla	123 ↗ id	fecha	A-Z mensaje
1	2	2025-10-21 17:46:48	Error General	
2	1	2025-10-21 17:43:25	Error de Deadlock (1213)	

- Comparar en la práctica 2 niveles de aislamiento (ej. READ COMMITTED y REPEATABLE READ) mostrando diferencias con ejemplos simples (basta

con un ejemplo breve que muestre una diferencia observable; no se requiere análisis de phantoms ni next-key locks).

Existen diferentes niveles de aislamiento en MySQL que se usan como mecanismo de control de concurrencia, evitando conflictos ante la ejecución de transacciones en forma simultánea.

El REPEATABLE READ es el que se usa por defecto en MySQL: No permite lectura sucia (una transacción no puede leer datos modificados por otra si no se confirmaron); la lectura no es repetible (evita obtener resultados distintos si un dato se lee dos veces y otra transacción lo modifica entre lecturas); y admite lectura fantasmas (cuando se consulta con una transacción nuevamente y aparecen nuevos registros ingresados por otra transacción al repetir la consulta).

A diferencia de REPEATABLE READ, el READ COMMITTED si puede generar distintos resultados al leer un mismo dato dos veces, si entremedio hay una transacción que modifica el dato.

Por su parte el READ UNCOMMITTED permite la lectura sucia, la lectura es repetible y pueden existir lecturas fantasmas. Del lado contrario, el SERIALIZABLE no permite la lectura sucia, la lectura no es repetible y no admite lecturas fantasmas.

Para poder desarrollar un ejemplo práctico y observable, debemos establecer el nivel de aislamiento y ejecutar las transacciones correspondientes en las sesiones que debemos abrir simultáneamente.

Por ejemplo, podemos generar una lectura sucia (dirty read) modificando el nivel de aislamiento a READ UNCOMMITTED. Si una transacción de la sesión 1 modifica el área de trabajo de un empleado, la transacción de la sesión 2 lo lee antes del COMMIT de la sesión 1. Pero si la sesión 1 hace ROLLBACK, la transacción de la sesión 2 lee un dato que nunca existió.

Para poder observar este ejemplo, ejecutamos el siguiente script (ver 09_concurrencia_guiada.sql):

```
● ● ●

-- Sesión 1
USE empleados;
SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;
START TRANSACTION;
UPDATE empleado SET area = 'Marketing' WHERE id = 2;

-- Sesión 2
USE empleados;
SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;
START TRANSACTION;
SELECT * FROM empleado WHERE id = 2;

-- Sesión 1
ROLLBACK;
SELECT * FROM empleado WHERE id = 2;

-- Sesión 2
SELECT * FROM empleado WHERE id = 2;
```

Sesión 1:

```
> USE empleados

0 row(s) modified.

> SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED

0 row(s) modified.

> START TRANSACTION

0 row(s) modified.

> UPDATE empleado SET area = 'Marketing' WHERE id = 2

1 row(s) modified.

> ROLLBACK

0 row(s) modified.

> SELECT * FROM empleado WHERE id = 2

id|eliminado|nombre|apellido|dni      |email           |fechaIngreso|area      |
---+-----+-----+-----+-----+-----+-----+-----+
 2|      0|Camila|Pérez   |730563892|micaela.lópez2@empresa.com| 2024-02-25|Tecnología|
1 row(s) fetched.
```

Sesión 2:

```
> USE empleados

0 row(s) modified.

> SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED

0 row(s) modified.

> START TRANSACTION

0 row(s) modified.

> SELECT * FROM empleado WHERE id = 2

id|eliminado|nombre|apellido|dni      |email           |fechaIngreso|area      |
---+-----+-----+-----+-----+-----+-----+-----+
 2|      0|Camila|Pérez   |730563892|micaela.lópez2@empresa.com| 2024-02-25|Marketing|
1 row(s) fetched.

> SELECT * FROM empleado WHERE id = 2

id|eliminado|nombre|apellido|dni      |email           |fechaIngreso|area      |
---+-----+-----+-----+-----+-----+-----+-----+
 2|      0|Camila|Pérez   |730563892|micaela.lópez2@empresa.com| 2024-02-25|Tecnología|
1 row(s) fetched.
```

De esta forma, podemos ver como la sesión 2 hace una lectura sucia de un dato que ha sido modificado por la transacción de la sesión 1 pero que no ha sido confirmado con un COMMIT. Una vez hecho el ROLLBACK, vuelve a leer el dato que realmente se encontraba guardado.

Ahora repetimos el proceso con SERIALIZABLE:

```
-- Sesión 1
USE empleados;
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
START TRANSACTION;
UPDATE empleado SET area = 'Marketing' WHERE id = 2;
SELECT * FROM empleado WHERE id = 2;

-- Sesión 2
USE empleados;
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
START TRANSACTION;
SELECT * FROM empleado WHERE id = 2;
```

Sesión 1:

```
> SET TRANSACTION ISOLATION LEVEL SERIALIZABLE
0 row(s) modified.

> START TRANSACTION
0 row(s) modified.

> UPDATE empleado SET area = 'Marketing' WHERE id = 2
1 row(s) modified.

> SELECT * FROM empleado WHERE id = 2
id|eliminado|nombre|apellido|dni      |email          |fechaIngreso|area      |
--+-----+-----+-----+-----+-----+-----+-----+
2|        0|Camila|Pérez    |730563892|micaela.lópez2@empresa.com| 2024-02-25|Marketing|
1 row(s) fetched.
```

Sesión 2:

```
> SET TRANSACTION ISOLATION LEVEL SERIALIZABLE
0 row(s) modified.

> START TRANSACTION
0 row(s) modified.

> SELECT * FROM empleado WHERE id = 2
id|eliminado|nombre|apellido|dni      |email           |fechaIngreso|area      |
---+-----+-----+-----+-----+-----+-----+-----+
 2|      0|Camila|Pérez    |730563892|micaela.lópez2@empresa.com| 2024-02-25|Tecnología|
1 row(s) fetched.
```

En este caso podemos observar que en la sesión 1 se actualiza el empleado (antes de hacer COMMIT o ROLLBACK), mientras que en la sesión 2, no se hace una lectura sucia, ya que se mantiene el área de trabajo en “Tecnología”.

Observaciones sobre concurrencias y transacciones

En esta etapa se comprobó que la concurrencia es uno de los desafíos más importantes en la gestión de datos. La simulación manual de deadlock (Error 1213) que realizamos demostró que, sin un control adecuado, las sesiones simultáneas pueden bloquearse mutuamente al acceder a recursos, requiriendo que el motor tenga que detener una transacción.

La transacción (con START, COMMIT, ROLLBACK) es la herramienta para gestionar esto, garantizando la atomicidad de las operaciones. Se observó que un manejo robusto requiere capturar errores específicos (HANDLER FOR 1213), ejecutar un ROLLBACK inmediato para mantener la integridad de los datos, y registrar el incidente (logging) para auditoría.

El procedimiento almacenado implementado automatiza esta solución e incluye una lógica de reintento (retry), tratando el deadlock no como un error fatal, sino como un fallo temporal que puede resolverse automáticamente, aumentando la robustez del sistema.

Anexo I: Uso de IA

Se copia y pega un prompt inicial, obteniendo la siguiente respuesta:

¡Entendido! Asumo completamente mi rol como asistente guía para la materia **Base de Datos I**.

Mi objetivo es claro: ayudarte a razonar y a construir tu propio conocimiento, no darte las respuestas de forma directa. Seguiré al pie de la letra las instrucciones para que nuestra interacción sea una verdadera herramienta de aprendizaje.

En resumen, mi enfoque será:

- **Guiar, no resolver:** Te daré pistas y te haré preguntas para que vos mismo llegues a la conclusión correcta.
- **Explicar el "porqué":** No solo te diré qué podría estar mal, sino que te explicaré el concepto que hay detrás para que no vuelvas a cometer el mismo error.
- **Lenguaje accesible:** Hablaremos en un idioma claro, como si estuviéramos en clase, evitando tecnicismos innecesarios.
- **Aprendizaje activo:** La meta es que, al final, te sientas seguro de tus propias soluciones porque entendiste el proceso.

Estoy listo para empezar cuando quieras. Contame en qué parte del trabajo práctico estás, mostrame tu diagrama o la tabla sobre la que tenés dudas. ¡Vamos a resolverlo juntos!

Etapa 1 – Modelado y Definición de Constraints

I. Se pegó el script utilizado (ver 01_esquema.sql)

¡Hola! Muy buen trabajo con el código SQL. Está muy prolíjo, es claro y sintácticamente correcto. ¡Va a funcionar sin problemas! 

Me gusta mucho que hayas usado:

- AUTO_INCREMENT para las claves primarias.
- Restricciones NOT NULL y UNIQUE donde corresponde (¡muy bien ese DNI!).
- Una FOREIGN KEY para relacionar las dos tablas.
- Incluso un CHECK para validar el formato del email, ¡excelente detalle!

Ahora, salgamos un poco del código y pensemos en el diseño del modelo. Acá es donde se pone más interesante el análisis.

Creaste una tabla empleado y una tabla legajo. Miremos la relación que definiste entre ellas con esta línea en la tabla legajo:

SQL

empleado_id BIGINT UNIQUE NOT NULL

La FOREIGN KEY dice que un legajo pertenece a un empleado. La palabra UNIQUE agrega otra regla.

Mi pregunta para vos es: según esa configuración, ¿cuántos legajos puede tener un empleado? 🤔 Piénsalo un segundo y contame qué te parece.

II. Cada empleado puede tener un único legajo, la relación es 1:1

¡Exacto! Lo entendiste perfectamente. Es una relación uno a uno. Cada empleado tiene un solo legajo, y cada legajo corresponde sí o sí a un solo empleado.

Ahora viene la pregunta clave en el diseño de bases de datos: si dos "cosas" (entidades) están tan íntimamente conectadas, ¿realmente necesitamos tenerlas en dos tablas separadas?

Pensemos en un caso práctico:

Necesitás mostrar en una pantalla una lista con el nombre, apellido, DNI y el número de legajo de todos los empleados que están 'ACTIVOS'.

Con tu diseño actual, que es funcionalmente correcto, ¿qué operación SQL vas a tener que hacer siempre para poder juntar los datos de la tabla empleado con los de la tabla legajo?

III. Es útil tener las tablas separadas para organizar y dotar mayor seguridad a la relación. Para la consulta del caso práctico debemos utilizar los comandos SELECT, FROM y WHERE para las condiciones de búsqueda, además de JOIN para unir las tablas.

La respuesta sobre la consulta SQL es perfecta. Efectivamente, para obtener los datos combinados necesitarías usar un JOIN. ¡Bien identificado!

Ahora, analicemos tu justificación para mantenerlas separadas: "organizar y dotar mayor seguridad".

Seguridad

Este es un punto muy interesante y avanzado. ¡Es genial que lo estés pensando! Separar datos por seguridad es una práctica real y muy útil cuando una parte de la información es mucho más sensible que la otra. Por ejemplo, si tuvieras una tabla con datos de sueldo o informes médicos, separarla sería una decisión excelente.

Ahora, pensemos en tu caso: los datos en legajo son nroLegajo, categoría y estado. ¿Considerás que estos datos son mucho más sensibles que el nombre o el dni del empleado como para necesitar esa separación? 🤔

Organización y el Costo del JOIN

Identificaste que necesitás un JOIN. Esa es la clave de todo este análisis.
Vas a necesitar un JOIN para ver la lista de empleados.
Vas a necesitar un JOIN para ver el perfil completo de un solo empleado.
Vas-a-necesitar-un-JOIN-para-casi-todo.
Si la relación es 1 a 1 y en la práctica siempre vas a querer los datos juntos, mantenerlos separados te obliga a escribir consultas más complejas y le da un pequeño trabajo extra al motor de la base de datos cada vez.
La solución más común y eficiente en estos casos es fusionar las tablas.

Piensa en tu modelo si se viera así:

SQL

```
CREATE TABLE empleado (
    id BIGINT AUTO_INCREMENT PRIMARY KEY,
    eliminado BOOLEAN DEFAULT FALSE,
    -- Datos personales
    nombre VARCHAR(80) NOT NULL,
    apellido VARCHAR(80) NOT NULL,
    dni VARCHAR(15) NOT NULL UNIQUE,
    email VARCHAR(150),
    fechalngreso DATE,
    -- Datos del legajo (ahora parte del empleado)
    nroLegajo VARCHAR(20) NOT NULL UNIQUE,
    area VARCHAR(50),
    categoria VARCHAR(30),
    estado ENUM('ACTIVO', 'INACTIVO') NOT NULL,
    fechaAlta DATE,
    observaciones VARCHAR(255),
    CONSTRAINT chk_email_formato CHECK (email LIKE '%@%.%' OR email IS NULL)
);
```

Ventajas de este diseño:

- Simplicidad: Para ver toda la info de un empleado, solo hacés un SELECT a una tabla. No más JOINs.
- Rendimiento: Es ligeramente más rápido, ya que la base de datos no necesita hacer la operación de unión.
- Lógica: Todos los atributos que describen a una única entidad (el empleado) están en un único lugar.

Separar en una relación 1 a 1 se justifica en casos muy específicos, pero como regla general, si son inseparables en la práctica, mantenlos juntos en tu modelo. ¿Qué te parece esta idea?

IV. Para probar la estructura de la base de datos, necesito aplicar validaciones que sean válidas e inválidas. Ya conozco la estructura, necesito generar valores aleatorios usando INSERT INTO y VALUES.

Claro, aquí tienes ejemplos de sentencias INSERT tanto válidas como inválidas para que puedas probar la estructura y las restricciones de tus tablas empleado y legajo.

Inserciones Válidas

Estos INSERT deberían funcionar sin problemas porque cumplen con todas las reglas que definiste (NOT NULL, UNIQUE, CHECK, FOREIGN KEY).

Paso 1: Insertar Empleados

Primero, necesitamos crear algunos empleados para que sus id puedan ser referenciados en la tabla legajo.

SQL

```
-- Insertar el primer empleado (suponiendo que el ID autogenerado será 1)
INSERT INTO empleado (nombre, apellido, dni, email, fechalIngreso, area)
VALUES ('Juan', 'Pérez', '30123456', 'juan.perez@example.com', '2022-05-10', 'Ventas');

-- Insertar el segundo empleado (suponiendo que el ID autogenerado será 2)
INSERT INTO empleado (nombre, apellido, dni, email, fechalIngreso, area)
VALUES ('Ana', 'Gómez', '32789012', 'ana.gomez@example.com', '2023-01-15', 'RR.HH.);

-- Insertar un empleado sin email (permitido porque es nullable)
INSERT INTO empleado (nombre, apellido, dni, fechalIngreso, area)
VALUES ('Carlos', 'López', '34555666', '2021-11-20', 'Sistemas');
```

Paso 2: Insertar Legajos (Asociados a los empleados)

Ahora creamos los legajos usando los id de los empleados que acabamos de insertar.
Recuerda que la relación es 1:1.

SQL

```
-- Legajo para Juan Pérez (ID 1)
INSERT INTO legajo (id, nroLegajo, categoria, estado, fechaAlta)
VALUES (1, 'LEG-2022-001', 'Vendedor Jr.', 'ACTIVO', '2022-05-10);

-- Legajo para Ana Gómez (ID 2)
INSERT INTO legajo (id, nroLegajo, categoria, estado, fechaAlta, observaciones)
VALUES (2, 'LEG-2023-002', 'Analista', 'ACTIVO', '2023-01-15', 'Periodo de prueba finalizado.');
```

Inserciones Inválidas

Estos INSERT fallarán a propósito, y cada uno te mostrará cómo actúan tus restricciones.

1. Violación de UNIQUE en dni (Empleado)

Intentar insertar un empleado con un DNI que ya existe.

SQL

```
-- Esto fallará porque el DNI '30123456' ya pertenece a Juan Pérez.
INSERT INTO empleado (nombre, apellido, dni, email, fechalIngreso, area)
VALUES ('Roberto', 'García', '30123456', 'roberto.g@example.com', '2024-02-01', 'Ventas');
```

2. Violación de NOT NULL en apellido (Empleado)

Intentar insertar un empleado sin un apellido, que es un campo obligatorio.

SQL

-- Esto fallará porque 'apellido' no puede ser nulo.

```
INSERT INTO empleado (nombre, dni, email, fechalngreso, area)
```

```
VALUES ('Mariana', '36111222', 'mariana.m@example.com', '2024-03-03', 'Marketing');
```

3. Violación del CHECK de formato de email (Empleado)

Intentar insertar un email sin "@" o sin un punto después.

SQL

-- Esto fallará porque el formato del email no es válido.

```
INSERT INTO empleado (nombre, apellido, dni, email, fechalngreso, area)
```

```
VALUES ('Pedro', 'Martínez', '29888777', 'pedro_martinez.com', '2020-10-10', 'Soporte');
```

4. Violación de FOREIGN KEY (Legajo)

Intentar crear un legajo para un empleado_id que no existe.

SQL

-- Esto fallará porque no existe ningún empleado con ID 99.

```
INSERT INTO legajo (id, nroLegajo, categoria, estado, fechaAlta)
```

```
VALUES (99, 'LEG-2024-099', 'Gerente', 'ACTIVO', '2024-01-01');
```

5. Violación de UNIQUE en la clave primaria de legajo (Relación 1:1)

Intentar crear un segundo legajo para un empleado que ya tiene uno.

SQL

-- Esto fallará porque ya existe un legajo para el empleado con ID 1 (Juan Pérez).

```
INSERT INTO legajo (id, nroLegajo, categoria, estado, fechaAlta)
```

```
VALUES (1, 'LEG-2024-003', 'Vendedor Sr.', 'ACTIVO', '2024-04-01');
```

Con estos ejemplos, podrás probar de forma práctica cada una de las reglas que has definido en tu base de datos. ¡Mucha suerte!

V. En base al script, ¿Los dominios de atributos son apropiados?

Sí, en general, los dominios de atributos que elegiste son muy apropiados para un diseño de este tipo. Demuestran una buena comprensión de los tipos de datos y sus usos.

Vamos a analizarlo en dos partes: los grandes aciertos y algunos puntos que, si bien no son errores, se podrían refinar para un diseño aún más robusto.

Puntos Fuertes (¡Muy bien hecho! )

1. Uso de VARCHAR para identificadores: Elegir VARCHAR para dni y nroLegajo es una decisión excelente. Mucha gente comete el error de usar tipos numéricos (INT), pero VARCHAR es superior porque:

- Maneja identificadores que pueden incluir letras, guiones o símbolos (ej: "LEG-1001").
 - Conserva los ceros a la izquierda, que son comunes en algunos sistemas de numeración.
 - Evita que se realicen operaciones matemáticas sin sentido sobre ellos (no tiene lógica sumar dos DNI).
2. Uso de DATE para fechas: Utilizar el tipo DATE es perfecto. Almacena la fecha en un formato estandarizado (año-mes-día) y es mucho más eficiente y seguro que guardarla como texto (VARCHAR).
 3. Uso de ENUM para estado: ¡Un acierto total! Usar ENUM('ACTIVO', 'INACTIVO') es la mejor opción aquí. Garantiza que solo esos dos valores puedan existir, eliminando cualquier posibilidad de errores de tipeo (como "Activo" con mayúscula o "Inactiva" en femenino). Aporta muchísima integridad de datos.

Puntos para Analizar (Posibles Mejoras 😊)

Aquí no hay errores, sino oportunidades de mejora o decisiones de diseño que vale la pena pensar.

1. id BIGINT vs. INT:
 - BIGINT puede almacenar un número astronómicamente grande (hasta 9 trillones). Para una tabla de empleados, un INT regular (que llega hasta 2.1 mil millones) suele ser más que suficiente y ocupa la mitad de espacio en disco.
 - ¿Está mal usar BIGINT? Para nada. Es una forma de asegurarte de que jamás te quedarás sin IDs, pero es bueno conocer la diferencia y elegir según la escala que esperas para tu sistema.
2. area y categoria como VARCHAR:
 - Esto funciona bien, pero tiene un riesgo: la inconsistencia por errores de tipeo. Un usuario podría ingresar "Tecnología" y otro "Tecnologia" (sin tilde), y para la base de datos serían dos áreas completamente distintas.
 - Alternativa A (La más profesional - Normalización): Crear dos tablas nuevas, una t_areas y otra t_categorias. En la tabla empleado, en lugar de guardar el nombre del área, guardarías un id_area (una Clave Foránea o Foreign Key) que apunte a la tabla t_areas.
 - Ventaja: Aseguras la consistencia total y si el nombre de un área cambia (ej: "Tecnología" pasa a llamarse "IT y Desarrollo"), solo lo modificas en un lugar.
 - Alternativa B (La más simple): Usar ENUM, tal como hiciste con estado. Si sabes que las áreas y categorías son pocas y no van a cambiar mucho (ej: ENUM('Tecnología', 'Recursos Humanos', 'Contabilidad')), esta es una solución rápida y efectiva para evitar los errores de tipeo.

En resumen: tu diseño actual es sólido, lógico y funcional. Las mejoras propuestas son detalles de optimización y buenas prácticas que te ayudarán a crear modelos aún más robustos y fáciles de mantener a futuro. ¡Excelente trabajo!

VI. Revisar si hay redundancias o inconsistencias.

Sí, en tu modelo actual hay dos puntos clave que pueden considerarse redundancias y que podrían llevar a inconsistencias lógicas. ¡Detectar esto es una parte fundamental del diseño de bases de datos!

1. fechalIngreso y fechaAlta

Actualmente tenés dos campos de tipo DATE: fechalIngreso y fechaAlta.

Esto es una redundancia porque es muy probable que ambos campos almacenen siempre la misma información. La fecha en que un empleado ingresa a la empresa suele ser la misma fecha en que se da de alta su registro en el sistema.

- El Riesgo: Si guardás el mismo dato en dos lugares, corrés el riesgo de que en el futuro alguien actualice uno y se olvide del otro, creando una inconsistencia. Por ejemplo, un empleado con fechalIngreso '2023-05-10' y fechaAlta '2023-05-11'. ¿Cuál es la correcta?
- Pregunta Guía: En el contexto de tu sistema, ¿existe algún caso real donde la fecha de ingreso de un empleado sea diferente a su fecha de alta?
- Solución Propuesta: A menos que tengas un motivo de negocio muy específico para separarlas, lo ideal es unificarlas en un solo campo. Podrías llamarlo fecha_ingreso o fecha_alta y eliminar el otro. Esto simplifica la tabla y elimina el riesgo de inconsistencia.

2. estado y eliminado

Este es un caso más sutil de redundancia conceptual que puede generar inconsistencias lógicas.

- estado ENUM('ACTIVO', 'INACTIVO'): Define el estado laboral del empleado. Un empleado 'INACTIVO' es alguien que ya no trabaja en la empresa (renunció, fue despedido, etc.).
- eliminado BOOLEAN: Es una bandera para un "borrado lógico" (soft delete), que usualmente se usa para ocultar un registro que ya no se utiliza o que fue creado por error.

Como ves, ambos campos apuntan a una idea similar: un empleado que ya no está operativo en el sistema.

- El Riesgo: Se pueden producir combinaciones confusas o contradictorias.
 - ¿Qué significa un empleado con estado = 'ACTIVO' y eliminado = TRUE? Es una inconsistencia lógica. Un empleado no puede estar activo y eliminado al mismo tiempo.
 - ¿Cuál es la diferencia entre un empleado estado = 'INACTIVO' y uno eliminado = TRUE? La lógica de tu aplicación tendría que ser muy cuidadosa para manejar ambos casos, que en la práctica son muy parecidos.
- Solución Propuesta: Unificar estos dos conceptos en el campo estado, haciéndolo más descriptivo. Podrías eliminar la columna eliminado y mejorar tu ENUM:
- SQL estado ENUM('ACTIVO', 'EN_LICENCIA', 'RENUNCIO', 'DESPEDIDO', 'JUBILADO')

- De esta forma, el campo estado te da una visión mucho más clara y precisa de la situación del empleado, sin necesidad de una bandera extra y evitando cualquier posible contradicción.

Eliminar estas redundancias hará tu diseño más limpio, robusto y fácil de mantener.

Etapa 2 – Generación y carga de datos masivos con SQL puro

- I. Necesito generar datos ficticios en gran volumen usando exclusivamente mecanismos de SQL y escalar la base de datos a un volumen de entre 200.000 y 500.000 registros totales. Entre los mecanismos posibles: Tablas "semilla" y producto cartesiano controlado; series/secuencias para enumerar y derivar atributos técnicos; tablas calendario para generar fechas/horas de forma sistemática; catálogos/maestras con cuotas por categoría/estado/rol; seudoaleatoriedad con sesgos (para evitar distribuciones uniformes); y expansiones iterativas o recursivas. ¿Cuál recomendarías para mi base de datos de empleados y legajos con una relación 1:1?

Para tu caso específico, con una base de datos de empleados y legajos en relación 1:1, la estrategia más efectiva y escalable es una combinación de tablas "semilla", un generador de series numéricas y el uso de aleatoriedad controlada con el operador módulo (%) o funciones RAND().

Esta combinación es ideal porque te permite crear un volumen masivo de datos variados y realistas a partir de un conjunto muy pequeño de datos iniciales, dándote control total sobre la cantidad y la distribución.

Plan de Acción Recomendado

Aquí te detallo un paso a paso para implementar esta estrategia y generar, por ejemplo, 250,000 empleados (lo que resultaría en 500,000 registros totales al sumar los legajos).

Paso 1: Crear tus Tablas "Semilla"

Estas son tablas pequeñas que contienen los componentes básicos para construir tus datos. La idea es que tengan suficientes valores para que las combinaciones parezcan únicas.

- _semilla_nombres: Una lista de nombres comunes.
- _semilla_apellidos: Una lista de apellidos.
- _semilla_cargos: Cargos o puestos de trabajo.
- _semilla_departamentos: Los departamentos de la empresa.

Ejemplo de estructura:

SQL

```
CREATE TABLE _semilla_nombres (
    id SERIAL PRIMARY KEY,
```

```
    nombre VARCHAR(50)
);
INSERT INTO _semilla_nombres (nombre) VALUES ('Juan'), ('María'), ('Carlos'), ('Ana'), ...;
-- Carga unos 50-100 nombres

CREATE TABLE _semilla_apellidos (
    id SERIAL PRIMARY KEY,
    apellido VARCHAR(50)
);
INSERT INTO _semilla_apellidos (apellido) VALUES ('García'), ('Rodríguez'), ('Pérez'),
('Fernández'), ...; -- Carga unos 50-100 apellidos
```

Tip clave: Usa un id numérico secuencial en cada tabla semilla. Esto es fundamental para el siguiente paso.

Paso 2: El Motor de Generación (La Serie Numérica)

Necesitas una forma de "invocar" el proceso de creación 250,000 veces. La manera más eficiente es con una función que genere una serie de números.

- En PostgreSQL (la más sencilla):
- SQL

```
SELECT generate_series(1, 250000) as id;
```

-
-
- En MySQL o SQL Server (usando una CTE recursiva):
- SQL

```
WITH RECURSIVE Numeros AS (
```

```
    SELECT 1 as id
    UNION ALL
    SELECT id + 1
    FROM Numeros
    WHERE id < 250000
```

```
)
```

```
SELECT id FROM Numeros;
```

-
-

Esta consulta te dará una tabla virtual con una columna id que va del 1 al 250,000. Esta será la base para crear cada empleado.

Paso 3: La Magia del JOIN Aleatorio

Ahora combinamos todo. Usaremos la tabla de números como base y la cruzaremos (JOIN) con las tablas semilla. El truco es no hacer un producto cartesiano completo, sino seleccionar un elemento "al azar" de cada tabla semilla para cada número de nuestra serie. La forma más simple de lograr esto es con el operador módulo (%).

SQL

```
-- Obtenemos el total de registros en cada tabla semilla
```

```
-- (Es mejor calcularlos una vez que hacerlo dentro de la consulta masiva)
DECLARE @total_nombres INT = (SELECT COUNT(*) FROM _semilla_nombres);
DECLARE @total_apellidos INT = (SELECT COUNT(*) FROM _semilla_apellidos);

-- ... y así para las otras tablas ...

-- Consulta de generación
INSERT INTO Empleados (nombre, apellido, fecha_nacimiento, id_departamento)
SELECT
    -- Selección "aleatoria" de nombre y apellido
    (SELECT nombre FROM _semilla_nombres WHERE id = (numeros.id % @total_nombres) + 1),
    (SELECT apellido FROM _semilla_apellidos WHERE id = (numeros.id % @total_apellidos) + 1),

    -- Generación de una fecha de nacimiento aleatoria (ejemplo para PostgreSQL)
    NOW() - '18 years'::interval - (RANDOM() * 30 * 365 || ' days')::interval,

    -- Asignación de departamento
    (numeros.id % @total_departamentos) + 1

FROM (SELECT generate_series(1, 250000) as id) AS numeros;
```

¿Cómo funciona (numeros.id % @total_nombres) + 1?

- numeros.id % @total_nombres devuelve el resto de la división entre el ID actual (de 1 a 250,000) y el total de nombres que tienes. El resultado siempre será un número entre 0 y @total_nombres - 1.
- Le sumamos 1 para que coincida con los id de tu tabla semilla, que van de 1 a @total_nombres.
- Esto crea un ciclo que asigna nombres y apellidos de forma distribuida. Para una aleatoriedad más "real", puedes usar (FLOOR(RANDOM() * @total_nombres) + 1).

Paso 4: Cargar la Tabla Dependiente (Legajos)

Como la relación es 1:1, una vez que la tabla Empleados está llena, puedes insertar en Legajos usando los id recién creados.

SQL

```
INSERT INTO Legajos (id_empleado, fecha_ingreso, salario_bruto)
SELECT
    id, -- El ID del empleado que acabamos de crear
    NOW() - (RANDOM() * 10 * 365 || ' days')::interval, -- Fecha de ingreso en los últimos 10 años
    400000 + (RANDOM() * 1500000) -- Salario aleatorio entre 400.000 y 1.900.000
FROM
    Empleados;
```

¿Por qué esta estrategia y no las otras?

- Tablas Calendario: Son excelentes para análisis, pero para generar fechas aleatorias, una función como NOW() - RANDOM() * '...' es más directa y no requiere una tabla extra.
- Expansiones Recursivas: Son muy potentes para datos jerárquicos (como un organigrama), pero para generar una lista "plana" de N registros, el generador de series es mucho más simple y rápido.
- Catálogos con Cuotas: ¡Este es el siguiente nivel de realismo! Puedes adaptar la estrategia que te di. Por ejemplo, en tu tabla _semilla_cargos, podrías añadir una columna probabilidad y usar una consulta más avanzada para que haya muchos "Analistas Junior" y pocos "Directores". Pero la base de la estrategia (semillas + generador de series) sigue siendo la misma.

En resumen, la combinación de semillas + series es el pilar fundamental para esta tarea. ¡Espero que te sirva para avanzar con tu trabajo práctico!

II. Nos convence utilizar solo el primer mecanismo que resulta simple y genera varias combinaciones de registros con las semillas.

¡Perfecto! Es una excelente elección. Esa estrategia es un pilar fundamental en la generación de datos de prueba porque es potente, controlable y fácil de entender.

Adelante. Mostrame cómo definirías tus primeras tablas "semilla" y qué datos se te ocurre ponerles. A partir de ahí, construimos la consulta de generación juntos.

III. Primero crear las tablas "semilla" nombres, apellidos, áreas y categorías con algunos valores insertados. Con este catálogo, deberíamos realizar la carga masiva mediante insert y alguna secuencia de números que sea aleatoria.

¡Excelente! Ese es exactamente el camino a seguir. Has creado tu "fábrica de datos" con las tablas semilla, y ahora vamos a ponerla a producir en masa.

La clave, como bien dices, es usar una única instrucción INSERT que se alimente de un SELECT. Este SELECT será nuestro generador inteligente, encargado de ensamblar los datos aleatorios.

Construyendo el INSERT Masivo

La estructura general de la consulta que necesitas es la siguiente:

SQL

INSERT INTO TuTablaPrincipal (columna1, columna2, ...)

SELECT

-- Aquí va la lógica para generar los datos de cada columna

FROM

-- Aquí va el "motor" que define cuántos registros crear

El Motor: La Secuencia de Números

Para generar, por ejemplo, 200,000 registros, necesitas algo que repita la acción 200,000 veces. La función generate_series es perfecta para esto.

FROM generate_series(1, 200000);

Esta línea crea una tabla virtual de una sola columna con 200,000 filas, asegurando que nuestro SELECT se ejecute esa cantidad de veces.

El Ensamblador: Selección Aleatoria de las Semillas

Ahora, por cada una de esas 200,000 filas, necesitamos tomar un valor al azar de cada tabla semilla. La forma más intuitiva y legible de hacer esto es con una subconsulta que ordene aleatoriamente y se quede con el primer resultado.

El patrón es: (SELECT columna FROM tabla_semilla ORDER BY RANDOM() LIMIT 1)

- ORDER BY RANDOM(): Baraja todas las filas de la tabla semilla.
- LIMIT 1: Se queda únicamente con la primera fila del resultado barajado.

Plantilla de la Consulta Final

Combinando todo, la consulta para tu carga masiva se vería así. Suponiendo que tu tabla principal se llama Empleados y tiene columnas para nombre, apellido, id_area e id_categoria:

SQL

INSERT INTO Empleados (nombre, apellido, id_area, id_categoria)

SELECT

-- Subconsulta para traer un nombre al azar de la semilla

(SELECT nombre FROM nombres ORDER BY RANDOM() LIMIT 1),

-- Subconsulta para traer un apellido al azar de la semilla

(SELECT apellido FROM apellidos ORDER BY RANDOM() LIMIT 1),

-- Subconsulta para traer un ID de área al azar

(SELECT id FROM areas ORDER BY RANDOM() LIMIT 1),

-- Subconsulta para traer un ID de categoría al azar

(SELECT id FROM categorias ORDER BY RANDOM() LIMIT 1)

FROM

-- Aquí defines cuántos registros quieres crear. ¡Ajusta el número como necesites!

generate_series(1, 200000);

Tu siguiente paso: Intenta adaptar esta plantilla a la estructura real de tu tabla principal. Muéstrame cómo te quedaría la consulta y la revisamos juntos para asegurar que todo esté en orden.

IV.

USE empleados;

INSERT INTO empleado (nombre, apellido, dni, email, fechalingreso, area)

```

WITH seq AS (
    SELECT (ROW_NUMBER() OVER ()) AS n
    FROM information_schema.columns a, information_schema.columns b
    LIMIT 400000 -- Límite de empleados a generar
)
SELECT
    (SELECT nombre FROM nombres_semilla ORDER BY RAND() LIMIT 1),
    (SELECT apellido FROM apellidos_semilla ORDER BY RAND() LIMIT 1),
    CONCAT(CAST(FLOOR(10000000 + RAND() * 90000000) AS CHAR), s.n),
    LOWER(CONCAT(
        (SELECT nombre FROM nombres_semilla ORDER BY RAND() LIMIT 1), '',
        (SELECT apellido FROM apellidos_semilla ORDER BY RAND() LIMIT 1),
        s.n,
        '@empresa.com'
    )),
    CURDATE() - INTERVAL FLOOR(RAND() * 3650) DAY,
    (SELECT area FROM areas_semilla ORDER BY RAND() LIMIT 1)
FROM
    seq s;
INSERT INTO legajo (nroLegajo, categoria, estado, fechaAlta, observaciones,
empleado_id)
SELECT
    CONCAT('LEG-', LPAD(e.id, 6, '0')),
    (SELECT categoria FROM categorias_semilla ORDER BY RAND() LIMIT 1),
    IF(RAND() < 0.9, 'ACTIVO', 'INACTIVO'),
    e.fechaIngreso,
    'Legajo creado automáticamente.',
    e.id
FROM
    empleado e
WHERE
    e.eliminado = FALSE;

```

¡Excelente trabajo! Este script es muy sólido y demuestra una comprensión clara de cómo generar datos en volumen. La forma en que separaste la carga de empleado y legajo es perfecta y el uso de la CTE para generar la secuencia es un método muy ingenioso en MySQL.

Vamos a revisar un par de detalles para pulirlo. No son errores graves, sino más bien oportunidades para hacer los datos un poco más consistentes y realistas.

INSERT en la tabla empleado

Tu consulta de INSERT para los empleados está muy bien estructurada. Tengo dos observaciones para que la consideres:

1. Generación del DNI

El código:

SQL

```
CONCAT(CAST(FLOOR(10000000 + RAND() * 90000000) AS CHAR), s.n)
```

Observación:

Aquí, estás generando un número aleatorio de 8 dígitos y luego le concatenas el número de secuencia s.n. Esto va a producir un DNI con más de 8 dígitos (por ejemplo, 45876123101), lo cual puede no ser válido. Además, aunque es poco probable, la parte aleatoria podría repetirse y solo s.n garantizaría la unicidad.

Sugerencia (más simple y segura):

Puedes generar un DNI único y con el formato correcto usando la secuencia s.n directamente. Por ejemplo, partiendo de un número base como 20000000:

SQL

```
(20000000 + s.n)
```

Con esta simple suma, te aseguras de que para las 400,000 filas que generas, cada DNI será único, correlativo y mantendrá siempre el formato de 8 dígitos.

2. Consistencia en el email

El código:

SQL

```
LOWER(CONCAT(
```

```
    (SELECT nombre FROM nombres_semilla ORDER BY RAND() LIMIT 1), '.',  
    (SELECT apellido FROM apellidos_semilla ORDER BY RAND() LIMIT 1), s.n,  
    '@empresa.com'
```

```
))
```

Observación:

¡Esto funciona perfecto! Pero hay un detalle conceptual interesante. El nombre y el apellido que usas para el email se "sortean" de nuevo, de forma independiente al nombre y apellido del empleado en esa misma fila. Por lo tanto, podrías tener un empleado llamado "Juan Pérez" con el email "ana.gomez123@empresa.com".

No es un error crítico y para un trabajo práctico es una solución totalmente válida. Lograr que coincidan requeriría una consulta más compleja (usando JOINs con las tablas semilla en lugar de subconsultas). Lo importante es que notes esta pequeña inconsistencia en los datos generados.

INSERT en la tabla legajo

SQL

```
INSERT INTO legajo (...) SELECT ... FROM empleado e WHERE e.eliminado = FALSE;
```

¡Esta parte está impecable! 🌟

- nroLegajo: La forma en que construyes el número de legajo ('LEG-000123') es profesional y muy clara.
- Estado con sesgo: El uso de IF(RAND() < 0.9, 'ACTIVO', 'INACTIVO') para que el 90% de los legajos estén activos es una técnica avanzada y genial para generar datos más realistas.
- Consistencia: Reutilizas el id y la fechalngreso del empleado, lo cual es perfecto para mantener la integridad de la relación 1:1.

Resumen

El script es de muy buena calidad. Mi recomendación principal sería ajustar la generación del DNI para que el formato sea el correcto. Con esa pequeña modificación, tu script de carga masiva estaría listo para ejecutarse.

¡Muy buen avance!

V. ¿Cómo podría probar la consistencia de los registros creados?

Para probar la consistencia de los registros que creaste, podés ejecutar una serie de consultas SQL. Cada consulta actúa como una "pregunta" que le hacés a tu base de datos para ver si los datos cumplen con las reglas que esperás.

Aquí te dejo varias pruebas agrupadas por el tipo de consistencia que verifican.

1. Chequeos de Unicidad (Uniqueness)

Estas pruebas aseguran que no haya duplicados en las columnas que deberían ser únicas.

- ¿Hay algún DNI repetido?
- SQL

```
SELECT dni, COUNT(*)  
FROM empleado  
GROUP BY dni  
HAVING COUNT(*) > 1;
```

- Resultado esperado: Esta consulta no debería devolver ninguna fila. Si aparece alguna, significa que tenés DNIs duplicados.

- ¿Hay algún email repetido?
- SQL

```
SELECT email, COUNT(*)  
FROM empleado  
GROUP BY email  
HAVING COUNT(*) > 1;
```

- Resultado esperado: Al igual que la anterior, debería estar vacía.

2. Chequeos de Integridad Referencial

Estas pruebas verifican que las relaciones entre tus tablas sean correctas.

- ¿Quedó algún empleado sin su legajo correspondiente? (Comprueba la relación 1:1)
- SQL

```
SELECT e.id, e.nombre, e.apellido
FROM empleado e
LEFT JOIN legajo l ON e.id = l.empleado_id
WHERE l.id IS NULL;
```

- Resultado esperado: La consulta no debería devolver filas. Si devuelve un empleado, significa que por alguna razón no se le creó su legajo.
- ¿Hay algún legajo que apunte a un empleado que no existe? (La prueba inversa)
- SQL

```
SELECT l.id, l.nroLegajo, l.empleado_id
FROM legajo l
LEFT JOIN empleado e ON l.empleado_id = e.id
WHERE e.id IS NULL;
```

- Resultado esperado: También debería estar vacía. Esto confirma que no hay "legajos huérfanos".

3. Chequeos de Lógica de Negocio

Estas pruebas validan que los datos tengan sentido según las reglas de una empresa real.

- ¿Son correctas las fechas de ingreso? (Por ejemplo, ninguna en el futuro)
- SQL

```
SELECT COUNT(*)
FROM empleado
WHERE fechalngreso > CURDATE();
```

- Resultado esperado: Debería devolver 0.
- ¿Se aplicó bien el sesgo de estado 'ACTIVO'/'INACTIVO'?
- SQL

```
SELECT
estado,
COUNT(*) AS total,
ROUND(COUNT(*) * 100.0 / (SELECT COUNT(*) FROM legajo), 2) AS porcentaje
FROM legajo
GROUP BY estado;
```

- Resultado esperado: Deberías ver dos filas. La de 'ACTIVO' debería tener un porcentaje cercano al 90% y la de 'INACTIVO' cercano al 10%, validando tu lógica con `RAND() < 0.9`.

4. Chequeo de Consistencia Interna

Esta es una prueba interesante para explorar los datos, como lo que hablamos sobre el email.

- ¿Cuántos emails no coinciden con el nombre y apellido del empleado?
- SQL

```
SELECT nombre, apellido, email
FROM empleado
WHERE email NOT LIKE LOWER(CONCAT(SUBSTRING_INDEX(nombre, ' ', 1), '.', apellido, '%'));
```

- Resultado esperado: Esta consulta sí devolverá muchos registros. No indica un error, sino que te permite observar el efecto de la generación aleatoria que usaste y confirma la inconsistencia que habíamos discutido.

Ejecutando estas consultas, tendrás una radiografía completa de la calidad y consistencia de tu carga masiva. ¡Es una práctica excelente para asegurar la calidad de tus datos!

Etapa 3 – Consultas Avanzadas y Reportes

- I. En esta etapa debemos desarrollar en base a las tablas, consultas que agreguen valor al sistema. Nuestro objetivo es pasar de consultas simples a consultas multi-tabla y agregadas, diseñar consultas que tengan utilidad práctica en el sistema, entender el impacto de la estructura de datos y los índices en consultas complejas. Metodología 1. Identificar áreas del CRUD donde un reporte agregue valor. 2. Diseñar consultas con JOIN, GROUP BY, HAVING, subconsultas y vistas. 3. Probar con grandes volúmenes de datos. 4. Medir tiempos con y sin índices en al menos 3 consultas representativas: Igualdad (p. ej., WHERE col = ? o IN (...)), • Rango (BETWEEN, >=, <=, LIKE 'pref%'), • JOIN (p. ej., A.fk = B.pk).

-Excelente

Tenés un punto muy interesante porque combina práctica SQL avanzada con análisis de rendimiento.

¿Tenés ya creadas las tablas en la base de datos? Si es así, ¿podrías decirme cuáles son y qué columnas principales tienen?

→ Con eso puedo ayudarte a pensar qué tipos de consultas complejas tendría sentido diseñar (por ejemplo, totales por mes, ranking de productos, combinaciones entre ventas y clientes, etc.).

¿Querés que empecemos por definir qué consultas podrían aportar valor al sistema o preferís arrancar directamente con una de las cuatro consultas obligatorias (JOIN, GROUP BY, subconsulta, vista)?

II. Se insertan las tablas ya plantadas de Empleado y Legajos.

-Perfecto.

Las tablas están muy claras y bien estructuradas: una tabla principal empleado y una tabla legajo que tiene una relación 1 a 1 con empleado (a través de empleado_id).

Vamos a pensar juntos cómo aprovechar estas tablas para cumplir con el punto del trabajo.

Algunas ideas posibles para tu caso:

1. JOIN útil → combinar empleado + legajo para mostrar empleados activos con su categoría.
2. GROUP BY + HAVING → agrupar por área y contar cuántos empleados hay en cada una.
3. Subconsulta → buscar empleados cuyo legajo tenga la fecha de alta más reciente.
4. Vista → crear una vista que reúna todos los datos relevantes de empleado y legajo, para simplificar reportes.

Pregunta para vos:

De estas ideas, ¿cuál te gustaría que trabajemos primero?

Podemos armarla paso a paso y luego la adaptamos para que cumpla con la consigna del trabajo (mediciones, EXPLAIN, índices, etc.).

III. Estaba pensando hacer las siguientes consultas:

- Muestra empleados con legajos inactivos
- Muestra los empleados junto con su legajo, categoría y estado
- Promedio de antigüedad (años) por área
- Lista empleados activos (según legajo) que trabajan en un área específica
- Área con más personal

Excelente selección 🎉

Tus ideas cubren todos los tipos de consultas que te piden en el trabajo, y además tienen sentido dentro del sistema.

Vamos a analizarlas una por una para que veas qué tipo de consulta representa cada una y cómo podrías encararlas paso a paso:

1. “Muestra empleados con legajos inactivos”

Tipo: JOIN + filtro (condición específica)

- Combina las dos tablas (empleado y legajo) usando el empleado_id.
- Luego filtrás donde estado = 'INACTIVO'.
 - 💡 Pista: usá INNER JOIN si querés solo los que tienen legajo, o LEFT JOIN si querés incluir empleados aunque no tengan uno (según tu caso).

2. “Muestra los empleados junto con su legajo, categoría y estado”

Tipo: JOIN completo (relación 1 a 1)

- Es una consulta básica de unión, ideal para crear una vista útil.
 - 💡 Pista: pensá en una vista llamada `vista_empleado_legajo` con las columnas más relevantes.
Así luego podés usarla para otros reportes sin escribir siempre el JOIN.

3. “Promedio de antigüedad (años) por área”

Tipo: GROUP BY + función agregada + cálculo de fechas

- Vas a agrupar por area y calcular la diferencia entre `fechaIngreso` y la fecha actual.
 - 💡 Pista: podés usar `TIMESTAMPDIFF(YEAR, fechaIngreso, CURDATE())`. Y luego aplicar `AVG()` sobre ese resultado.

4. “Lista empleados activos (según legajo) que trabajan en un área específica”

Tipo: JOIN + filtro + parámetro (igualdad)

- Muy buena para probar el impacto del índice en consultas por igualdad, porque tiene un WHERE por estado y area.
 - 💡 Pista: más adelante vas a poder probar con EXPLAIN y comparar con y sin índice en area.

5. “Área con más personal”

👉 Tipo: GROUP BY + ORDER BY + LIMIT 1 (subconsulta o HAVING)

- Ideal para practicar agregaciones y ranking de resultados.
 - 💡 Pista: podrías usar `COUNT(*)` para contar empleados por área y luego limitar al primero.

Lo pude realizar ahora quiero seguir con las vistas tenía pensada 2:

- Vista de empleados activos
- Vista de reportes generales para no mostrar datos sensibles de los empleados

Tus dos ideas de vistas son muy acertadas —una operativa y otra de consulta segura—, y además muestran que entendiste el sentido práctico del punto del trabajo (crear vistas que aporten valor al sistema).

Vamos a analizarlas paso a paso para que puedas definirlas correctamente:

1. Vista de empleados activos

👉 Objetivo: tener una vista rápida de todos los empleados cuyo legajo está “ACTIVO”. Esto evita tener que escribir el JOIN y el WHERE estado = 'ACTIVO' cada vez.



Pista:

Pensá qué columnas querés mostrar. Lo mínimo sería:

- nombre y apellido
- dni
- área
- categoría (desde legajo)
- estado

Con eso tendrías una vista práctica tipo “reporte de personal en actividad”.

Pregunta guía:

¿Querés que la vista muestre también la fecha de ingreso y el correo del empleado, o preferís solo los datos básicos?

2. Vista de reportes generales (sin datos sensibles)

👉 Objetivo: ofrecer una vista resumida para informes o consultas externas, sin exponer datos personales como dni o email.



Pista:

Podés incluir datos como:

- área
- categoría
- estado del legajo

- fecha de ingreso o antigüedad
Y excluir campos sensibles.

También podrías agregar columnas calculadas como la antigüedad (en años) o totales por área si lo ves útil.

IV. Necesito que me guíes en realizar 3 consultas de pruebas con gran cantidad de datos una consulta por igualdad otra por rango y con JOIN

Perfecto. Esa parte corresponde justo a la sección de “Medición comparativa con/sin índice”, donde tenés que probar tres tipos de consultas distintas y medir los tiempos de ejecución.

Vamos paso a paso.

Primero definimos qué tipo de condición representa cada prueba, y después te voy dando pistas para construir las consultas correctamente sobre tus tablas empleado y legajo.

Consulta por igualdad

Objetivo: probar el rendimiento cuando filtrás con un WHERE col = valor.

En tu caso, podrías usar algo así como buscar por área específica o DNI.

Pistas para diseñar:

Usá una columna que tenga valores repetidos, como área, para que el índice sea útil.

Consulta por rango

Objetivo: analizar el impacto de índices en comparaciones de tipo BETWEEN, >=, <=, o LIKE 'pref%'.

Pistas para tu caso:

Podrías usar fechalngreso o fechaAlta (de legajo).

Consulta con JOIN

Objetivo: medir cómo afecta un índice cuando hacés una combinación entre tablas.

Pistas:

- Tenés una relación 1 a 1 entre empleado.id y legajo.empleado_id.

Recomendación general para la prueba:

1. Insertá una gran cantidad de datos (pueden ser copias o datos generados automáticamente).
2. Ejecutá cada consulta 3 veces y anotá los tiempos (tomá la mediana).
3. Probá con y sin índices (DROP INDEX o IGNORE INDEX).
4. Usá EXPLAIN para observar cómo cambia el plan de ejecución.

V. Estamos tratando de generar una consulta áreas ordenadas por cantidad de empleados activos e inactivos, tenga tambien una columna con el porcentaje de empleados activos de cada area. te voy a plantear mi consulta base y poco a poco me vas a ayudar sin darme el resultado a optimizar y mejorarla.

Excelente elección de consulta: tiene JOIN, funciones de agregación, agrupamiento y cálculos derivados, así que es ideal para mostrar evolución y optimización paso a paso.

Hacelo así: Pasame tu consulta base tal como la tenés ahora, aunque no esté del todo bien o completa.

VI. **SELECT e.area, l.estado
FROM empleado e
JOIN legajo l ON e.id = l.empleado_id
ORDER BY e.area;**

Tu consulta base es un excelente punto de partida, porque ya lograste:

- unir correctamente las tablas con el JOIN,
- traer las columnas relevantes (área y estado),
- y ordenar el resultado por área

Ahora, pensemos juntos el siguiente paso lógico.

Para agrupar por área, necesitamos agregar un GROUP BY e.area.

Pero como tenés que contar según el estado, vas a necesitar alguna forma de contar condicionalmente.

Pregunta guía:

¿Recordás qué función o estructura en SQL te permite “contar solo si se cumple una condición”?

(Pista: suele usarse dentro de COUNT() o SUM() con un CASE WHEN ... THEN 1 ELSE 0 END.)

Intentá escribir la parte del conteo por área —por ejemplo, cantidad de activos por área— y te ayudo a revisarla.

VII. **Si así podría ser? :**

```
SELECT e.area, l.estado, COUNT(*) AS cantidad
FROM empleado e
JOIN legajo l ON e.id = l.empleado_id
GROUP BY e.area, l.estado
ORDER BY e.area;
```

Vas muy bien encaminado: ya diste el paso clave, que es agrupar por área y estado.

Pensá cómo podrías transformar las filas “ACTIVO / INACTIVO” en columnas separadas dentro de la misma fila.

Esto se logra con conteos condicionales

Entonces, intentá escribir una nueva versión de la consulta que:

- elimine l.estado del GROUP BY,
- y cuente activos e inactivos con CASE WHEN.

VIII. **SELECT e.area,**

```
SUM(CASE WHEN l.estado = 'ACTIVO' THEN 1 ELSE 0 END) AS
empleados_activos,
SUM(CASE WHEN l.estado = 'INACTIVO' THEN 1 ELSE 0 END) AS
empleados_inactivos
FROM empleado e JOIN legajo l ON e.id = l.empleado_id
GROUP BY e.area;
```

Ahora sí estás aplicando correctamente el conteo condicional.

Te explico por qué tu consulta está muy bien planteada, y te dejo una pista para el siguiente paso (el porcentaje).

Qué está perfecto:

- Usás SUM(CASE WHEN ... THEN 1 ELSE 0 END), que es la forma correcta de contar según una condición.
- Quitaste l.estado del GROUP BY, así que ahora cada área aparece una sola vez.
- El JOIN y la agrupación están bien.

Próximo paso: calcular el porcentaje

IX. **Queremos agregar una columna que indique el porcentaje de empleados activos por área, usando los dos valores que ya calculaste. Sería así?:**

```
SELECT
    e.area,
    COUNT(*) AS total_empleados,
    SUM(CASE WHEN l.estado = 'ACTIVO' THEN 1 ELSE 0 END) AS
    empleados_activos,
    SUM(CASE WHEN l.estado = 'INACTIVO' THEN 1 ELSE 0 END)
    AS empleados_inactivos,
    ROUND(
        SUM(CASE WHEN l.estado = 'ACTIVO' THEN 1 ELSE 0 END) /
    COUNT(*) * 100, 2
    ) AS porcentaje_activos
FROM empleado e
JOIN legajo l ON e.id = l.empleado_id
GROUP BY e.area
ORDER BY empleados_activos DESC, empleados_inactivos ASC;
```

Esa consulta ya está muy bien lograda: tenés todos los componentes que pide el trabajo y además está escrita de forma clara y ordenada.

Vamos a analizarla paso a paso, y después te dejo unas pistas para optimizarla o dejarla más prolja (como pediría un docente en la parte de “mejoras progresivas”).

Qué estás haciendo perfecto:

1. Agrupás correctamente por área.
→ GROUP BY e.area es lo que permite resumir los datos por departamento.
2. Mostrás el total y los conteos parciales.
→ COUNT(*) te da el total de empleados del área.
→ Los dos SUM(CASE WHEN ...) funcionan muy bien como conteos condicionales.
3. Calculás el porcentaje con precisión y formato.
→ Usás ROUND(..., 2) para limitar a dos decimales, lo cual es ideal.
→ Dividís la cantidad de activos por el total y multiplicás por 100.
4. Ordenás de forma lógica.
→ Mostrar primero las áreas con más empleados activos da valor analítico.

Pequeños detalles para mejorar (opcional, si querés documentar “optimizaciones”):

1. Evitar la división por cero (caso extremo):
Si algún área no tuviera empleados (algo raro, pero posible), la división podría fallar.
Podés prevenirlo con un NULLIF

Etapa 4 – Seguridad e Integridad

- I. La etapa Nº 4 del trabajo práctico que estoy realizando comprende la siguiente descripción: "Implementar medidas de seguridad en el acceso a datos y garantizar la integridad de la base. Esto implica aplicar mínimos privilegios, usar vistas para exponer datos, evitar SQL injection mediante consultas parametrizadas y demostrar que las restricciones de integridad se cumplen." Necesito que me guíes con la primera actividad: "Crear un usuario con privilegios mínimos y mostrar pruebas de acceso restringido."

¡Perfecto! Vamos a desglosar esa primera actividad. Es un pilar fundamental de la seguridad en bases de datos.

La tarea tiene dos partes principales:

1. Crear el usuario.
2. Demostrar que no puede hacer (casi) nada.

Parte 1: Creación del usuario

Para empezar, necesitamos saber con qué "herramienta" estamos trabajando, ya que el comando exacto puede variar.

¿Qué motor de base de datos estás usando para tu trabajo práctico? ¿Es MySQL, PostgreSQL, SQL Server u otro?

Sabiendo eso, puedo darte pistas más precisas sobre el comando que necesitas usar. 😊

MySQL y Dbeaver

Genial. Con MySQL, tienes dos formas de hacerlo:

1. Directamente con SQL: (Recomendado para entender qué pasa)
2. Con la interfaz gráfica de DBeaver: (Más visual)

Mi sugerencia es que intentes primero con el comando SQL, ya que te da control total y ves exactamente lo que estás ejecutando.

Pista 1: El comando SQL

El comando que buscas es CREATE USER. La sintaxis básica se ve así:

SQL

```
CREATE USER 'nombre_usuario'@'host' IDENTIFIED BY 'tu_contraseña_segura';
```

Analicemos esto:

- 'nombre_usuario': ¿Qué nombre quieres ponerle a este usuario? (Por ejemplo: 'usuario_limitado').
- 'host': Esto define desde dónde se puede conectar el usuario.
 - Usar 'localhost' es una buena práctica de seguridad, ya que solo permite conexiones desde la misma máquina donde está el servidor MySQL.
 - Usar '%' permitiría la conexión desde cualquier lugar, lo cual es menos seguro.

- 'tu_contraseña_segura': Elige una contraseña para él.

El punto clave: Cuando creas un usuario así, MySQL por defecto le da el privilegio USAGE. Este es el privilegio mínimo absoluto: le permite "iniciar sesión", pero no puede leer, escribir o modificar ninguna tabla de ninguna base de datos. Es como darle a alguien una llave de la puerta principal del edificio, pero no la llave de ningún apartamento.

Intenta escribir ese comando en un editor SQL de DBeaver y ejecutarlo.

Pista 2: Usando DBeaver (Alternativa)

Si prefieres la interfaz:

1. En el panel "Navegador de Base de Datos" (izquierda), expande tu conexión a MySQL.
 2. Busca la carpeta "Administración" o "Security".
 3. Haz clic derecho en "Usuarios" (o "Users") y selecciona "Crear nuevo usuario".
 4. DBeaver te abrirá una ventana donde puedes poner el nombre de usuario, el host (localhost) y la contraseña.
-

Tu siguiente paso:

Intenta crear el usuario (con el método SQL preferentemente). Una vez que lo tengas, ¿cómo se te ocurre que podríamos "demostrar" que realmente no tiene acceso a nada? 🤔

II. Primero deberíamos brindarle o poder observar que privilegios tiene ese usuario creado. Sin eso, no podemos demostrar nada (continúa con la pista 1)

¡Exacto! Muy bien pensado.

Ese es el paso lógico: primero verificamos lo que MySQL le asignó, y luego demostramos que esas asignaciones funcionan como esperamos.

Verificación de Privilegios

Para ver los privilegios que tiene tu nuevo usuario, puedes usar el comando SHOW GRANTS.

Suponiendo que creaste el usuario 'usuario_limitado'@'localhost', ejecutarías esto:
SQL

```
SHOW GRANTS FOR 'usuario_limitado'@'localhost';
```

¿Qué deberías ver?

Si todo salió bien, la salida será algo muy simple, parecido a esto:

```
GRANT USAGE ON *.* TO 'usuario_limitado'@'localhost'
```

Analicemos ese resultado:

- GRANT USAGE: Este es el privilegio del que hablamos. USAGE es literalmente el privilegio de "no tener privilegios". Solo le permite conectarse al servidor de MySQL.
- ON *.*: Esto significa que el privilegio (en este caso, USAGE) se aplica a "todas las bases de datos (*) y todas las tablas (*)".

Así que, GRANT USAGE ON *.* se traduce como: "Este usuario puede iniciar sesión, pero no tiene permiso para hacer absolutamente nada en ninguna tabla de ninguna base de datos".

Con esto, ya tienes la "prueba" teórica.

Parte 2: La Demostración Práctica

Ahora viene la parte divertida: demostrar que las restricciones se cumplen.

Si ya verificaste con SHOW GRANTS que solo tiene USAGE, ¿cómo se te ocurre que podrías demostrar en la práctica que no puede acceder a nada?

Pista: ¿Qué necesitarías hacer en DBeaver para "convertirte" en ese usuario y qué comandos intentarías ejecutar?

- III. Olvidá Dbeaver por el momento. Solo hablemos en términos de script .sql y consultas SQL. Yo diría que primero otorguemos algunos permisos, ya que justamente tenemos que crear un usuario con "privilegios mínimos" en la primer actividad, luego mostrar pruebas de acceso.**

Entendido. Nos enfocamos solo en SQL.

Tienes un punto muy válido. Un usuario que solo tiene USAGE (el permiso por defecto al crear) demuestra el extremo de la restricción (no puede hacer nada), pero en la práctica, "privilegios mínimos" casi siempre significa "lo mínimo necesario para hacer una tarea específica".

Tu interpretación es excelente para el trabajo práctico.

Escenario de Prueba

Imaginemos que en tu base de datos (vamos a llamarla tp_base_de_datos) tienes varias tablas, pero hay una en particular, digamos articulos, que este nuevo usuario necesita leer. El objetivo es: Darle permiso al 'usuario_limitado'@'localhost' para que pueda hacer solamente SELECT en la tabla articulos, y nada más.

Pista 1: Otorgando el Permiso

Para dar permisos, usamos el comando GRANT. La estructura básica que necesitas es:
SQL

GRANT [tipo_de_permiso] ON [dónde] TO [quién];

¿Cómo adaptarías esa plantilla para nuestro escenario?

- [tipo_de_permiso]: ¿Qué comando queremos permitirle? (SELECT, INSERT, UPDATE?)
- [dónde]: ¿Sobre qué objeto específico? (Necesitas indicar la base de datos y la tabla. La sintaxis es nombre_base_datos.nombre_tabla).
- [quién]: El usuario que creamos ('usuario_limitado'@'localhost').

Intenta escribir el comando GRANT completo.

Pista 2: La Demostración (Adelanto)

Una vez que ejecutes ese GRANT, necesitarás demostrar dos cosas para que la prueba sea completa:

1. Demostrar que puede hacer lo permitido: (Ej: ¿Funcionará un SELECT * FROM tp_base_de_datos.articulos;?)
2. Demostrar que NO puede hacer lo no permitido: (Ej: ¿Qué pasará si intenta un INSERT en articulos? ¿O un SELECT en otra tabla?)

Primero, concéntrate en escribir el comando GRANT correcto. Muéstrame cómo te quedaría.

Primero veamos como va mi asignación de privilegios hasta aquí:

-- Creación de usuario

CREATE USER 'tpi_bd'@'localhost' IDENTIFIED BY '1234';

-- Otorgando privilegios

GRANT SELECT, INSERT, UPDATE, DELETE ON empleados.empleado TO 'tpi_bd'@'localhost';

GRANT SELECT, INSERT, UPDATE, DELETE ON empleados.legajo TO 'tpi_bd'@'localhost';

IV. Creo que ya con esto puedo hacer las pruebas de acceso restringido, ya que estoy dandole acceso a este nuevo usuario para usar SELECT, INSERT, UPDATE, DELETE , pero no DROP, por ejemplo.

¡Vas muy bien encaminado!

Tus comandos CREATE USER y GRANT son sintácticamente correctos. Has creado un usuario y le has asignado permisos específicos sobre tablas concretas.

Tu razonamiento para las pruebas también es excelente:

1. Probar algo que sí debería funcionar (un SELECT o INSERT en empleados.empleado).
2. Probar algo que no debería funcionar (un DROP TABLE empleados.empleado).

Esto demuestra perfectamente que el usuario tpi_bd no es un administrador. ¡Buen trabajo!

Reflexión sobre "Mínimos Privilegios"

Ahora, un pequeño punto conceptual sobre "mínimos privilegios" para que tu trabajo práctico quede impecable.

Lo que hiciste está bien para demostrar una restricción (CRUD vs. DROP), pero piénsalo así: ¿SELECT, INSERT, UPDATE y DELETE (conocido como CRUD) son realmente los permisos mínimos?

Le has dado a este usuario el poder de borrar (DELETE) todos los registros de la tabla empleado. ¡Eso es un permiso muy alto y peligroso! 🚨

El "Principio de Mínimos Privilegios" dice que un usuario solo debe tener exactamente los permisos que necesita para su tarea, y nada más.

Preguntas para pensar:

- ¿El rol de este usuario (tpi_bd) es ser un administrador de datos que necesita poder borrar empleados?
- ¿O es un usuario de una aplicación que quizás solo necesita consultar (SELECT) y agregar (INSERT) nuevos empleados?

En un escenario real, un usuario de "solo consulta" tendría únicamente SELECT. Un usuario de "entrada de datos" quizás tendría SELECT e INSERT, pero rara vez DELETE o UPDATE sobre toda la tabla.

Para tu TP, lo que hiciste es válido, pero si quisieras ser súper estricto con el concepto, podrías darle aún menos permisos (por ejemplo, solo SELECT).

Tu Siguiente Paso: Las Pruebas

Ahora, ¡a probar! Tu plan es bueno. Te falta solo una prueba crucial para que la demostración sea completa.

Para demostrar que el acceso está realmente restringido, necesitas probar 3 cosas:

1. Que puede hacer lo permitido:
 - (Ej: SELECT * FROM empleados.empleado;) -> Debería funcionar.
2. Que NO puede hacer algo que requiere más permisos en ESA tabla:
 - (Ej: DROP TABLE empleados.empleado;) -> Debería fallar. (¡Tu idea!)
3. Que NO puede hacer NADA en otras tablas:
 - Esta es la prueba que te falta. ¿Qué pasa si tu base de datos empleados también tiene una tabla llamada departamentos?

¿Qué comando intentarías ejecutar contra esa tabla departamentos para probar que tpi_bd no tiene acceso a ella?

V. **Bueno, entonces dado lo que mencionas quitaría DELETE de los "permisos mínimos". Respecto a la tabla "departamentos", que de momento no existe, lo que sucederá es que el nuevo usuario no puede acceder a ella, por lo tanto podría ser otra prueba para observar qué es lo que no tiene permitido el usuario.**

¡Buenísimo! Esa es exactamente la mentalidad correcta para "mínimos privilegios".

1. Quitar DELETE: Es una decisión muy acertada. Limita el daño potencial que ese usuario puede hacer. Un usuario que puede hacer SELECT, INSERT y UPDATE tiene un rol claro (gestión de datos), pero no puede hacer borrados masivos.
2. Probar otra tabla (departamentos): ¡Exacto! Esa es la prueba crucial. Demuestra que los permisos que diste son específicos para las tablas empleado y legajo, y no para toda la base de datos empleados.

Tu Siguiente Paso: Armar el "Script de Prueba"

Ahora estás listo para la demostración. Lo que necesitas es un conjunto de consultas SQL que demuestren las dos caras de la moneda: lo que tpi_bd puede hacer y lo que no puede hacer.

Imagina que "cierras sesión" como administrador (root) y "abres sesión" como tpi_bd (en DBBeaver crearías una nueva conexión usando ese usuario; en SQL puro, te conectarías con él).

Escribe las consultas que intentarías ejecutar para probar tus restricciones. Deberías incluir al menos 3 pruebas:

1. Prueba de éxito: Una consulta que debería funcionar (basada en los GRANT que SÍ diste).
2. Prueba de permiso denegado (en tabla permitida): Una consulta que debería fallar en las tablas empleado o legajo (como el DELETE que quitaste).
3. Prueba de acceso denegado (en tabla no permitida): La consulta contra la tabla departamentos (incluso si no existe, el error que dará MySQL será "Access denied", que es lo que quieras demostrar).

¿Cuáles serían esas 3 consultas SQL que escribirías para tu demostración?

VI. Ya tengo las primeras dos, pero primero deberíamos hacer una observación sobre lo siguiente. La tabla departamentos no existe en mi base de datos, por lo que no estaríamos probando exactamente que un usuario no puede acceder a ella

¡Excelente observación! Tienes toda la razón. Es una distinción conceptual muy importante. Si intentas consultar una tabla que no existe (departamentos), el error que te dará MySQL será algo como:

ERROR 1146 (42S02): Table 'empleados.departamentos' doesn't exist

Este error ocurre antes de que MySQL revise los permisos. No estás probando una falla de seguridad, estás probando que la tabla no existe.

Para que tu prueba sea válida y demuestre restricción de acceso, necesitas intentar acceder a un objeto que sí exista, pero para el cual no tengas permisos.

Pista: La prueba definitiva

No necesitas crear una tabla nueva. Hay tablas en el sistema que siempre existen y a las que un usuario de bajos privilegios nunca debería acceder.

La base de datos mysql es el corazón del servidor: almacena las tablas de usuarios, los permisos, etc. Intentar leerla es la prueba de seguridad perfecta.

Por ejemplo, la tabla mysql.user almacena todos los usuarios y sus contraseñas (hasheadas).

Tu objetivo: Demostrar que tpi_bd no puede "espiar" la lista de usuarios del sistema.

Tu Siguiente Paso

Ahora sí, ¿cuáles serían las 3 consultas SQL (como el usuario tpi_bd) que ejecutarías para tu demostración?

1. Prueba de éxito: (Una que SÍ funciona en empleados.empleado).

2. Prueba de permiso denegado: (Una que NO funciona en empleados.empleado, como DELETE o DROP).
3. Prueba de acceso denegado (en otra DB): (La consulta para intentar leer la tabla user de la base de datos mysql).

Escribe esas tres consultas.

Dicho eso, en base a tu último comentario, mi script de pruebas hasta el momento y para cada caso es el siguiente:

1.
USE empleados;

SELECT * FROM empleado LIMIT 10;

2. USE empleados;

DROP TABLE legajo;

3. SELECT * FROM mysql.user;

¡Excelente! Ese script de prueba es perfecto.

Vamos a analizar por qué cada prueba es ideal para tu demostración:

Prueba 1: Éxito (Lo que SÍ puede hacer)

SQL

USE empleados;

SELECT * FROM empleado LIMIT 10;

- Propósito: Demostrar que el GRANT SELECT ... ON empleados.empleado que le diste, funcionó.
- Resultado esperado: El usuario tpi_bd podrá ver las primeras 10 filas de la tabla empleado. 

Prueba 2: Permiso denegado (En tabla permitida)

SQL

USE empleados;

DROP TABLE legajo;

- Propósito: Demostrar que, aunque tiene permisos CRUD (o casi CRUD) sobre legajo, no es un administrador. No tiene el permiso DROP.
- Resultado esperado: Falla. MySQL devolverá un error del tipo DROP command denied to user 'tpi_bd'@'localhost' for table 'legajo'. 

Prueba 3: Acceso denegado (En base de datos no permitida)

SQL

```
SELECT * FROM mysql.user;
```

- Propósito: Esta es la prueba clave de "mínimos privilegios". Demuestra que el usuario no puede "curiosear" en otras partes del sistema, especialmente en la base de datos mysql que guarda la seguridad.
- Resultado esperado: Falla. MySQL devolverá un error del tipo SELECT command denied to user 'tpi_bd'@'localhost' for table 'user'. 

Con estos tres scripts y sus resultados (uno exitoso, dos fallidos por las razones correctas), has demostrado de manera contundente la creación de un usuario con privilegios mínimos y has mostrado pruebas claras de su acceso restringido.

Has completado esta primera actividad. ¿Cuál es el siguiente punto de tu trabajo práctico?

VII. El siguiente punto es: Diseñar 2 vistas que oculten información sensible. Este es el esquema de mi base de datos (ver en 01_esquema.sql):

¡Excelente! Las Vistas (VIEW) son una de las mejores herramientas para este punto de la consigna.

Una Vista es, en esencia, una consulta SELECT que se guarda y se le da un nombre. Se comporta como una "tabla virtual".

Su gran ventaja para la seguridad es que puedes usarla para crear una "ventana" a tus datos, mostrando solo lo que tú quieras que se vea. Puedes:

1. Ocultar Columnas: (Seguridad a nivel de columna).
2. Ocultar Filas: (Seguridad a nivel de fila).

Pista 1: Identificando lo "Sensible"

Antes de escribir código, miremos tu esquema. ¿Qué columnas en empleado y legajo consideras "sensibles" o privadas? ¿Qué información no debería ver un usuario de consulta común?

- En empleado: ¿Quizás el dni o el email?
- En legajo: ¿Quizás observaciones (que podría tener notas privadas) o el empleado_id (que es un dato interno)?

Pista 2: Creando las Vistas

La tarea te pide dos vistas. Pensemos en dos escenarios de uso:

Escenario A: Un "Directorio de Personal"

- Propósito: Una vista para que cualquier empleado pueda ver quién trabaja en qué área.
- ¿Qué mostraría? Quizás nombre, apellido, area.
- ¿Qué ocultaría? Definitivamente id, dni, email, fechalngreso, eliminado.

Escenario B: Una "Vista de Legajos Activos" (para un gerente)

- Propósito: Un gerente que necesita ver el estado de sus empleados, pero sin ver notas confidenciales.

- ¿Qué mostraría? Quizás el nroLegajo, categoria, estado, y el nombre/apellido del empleado (¡esto requeriría un JOIN!).
 - ¿Qué ocultaría? La columna observaciones.
 - ¿Qué filas filtraríamos? Probablemente solo queremos mostrar los que tienen estado = 'ACTIVO' y eliminado = FALSE.
-

Tu Siguiente Paso

La sintaxis básica para crear una vista es:

SQL

```
CREATE VIEW nombre_de_la_vista AS
SELECT ...
FROM ...
WHERE ...;
```

Intenta escribir la consulta SQL para la Vista 1 (Escenario A). ¿Cómo armarías ese SELECT para que solo muestre el nombre, apellido y área de los empleados que no estén eliminados?

VIII. En base a tus sugerencias, dado que ya habíamos hecho una vista pública con ciertas restricciones (le agregamos eliminado = FALSE, que no lo tenía), utilizaremos esa en el escenario A y sí realizamos una vista en base a tu sugerencia en el escenario B. Estas son las vistas hasta el momento:

-- VISTAS

-- VISTA 1: vista_empleados_publica ya creada en la etapa 3, solo le agregamos esta vez `eliminado = FALSE`

```
CREATE OR REPLACE VIEW vista_empleados_publica AS
SELECT
    e.nombre,
    e.apellido,
    e.area,
    l.categoría,
    l.estado
FROM
    empleado AS e
JOIN
    legajo AS l ON e.id = l.empleado_id
WHERE
    e.eliminado = FALSE
    AND l.eliminado = FALSE;
```

**-- VISTA 2: Vista gerencial de legajos denominada `vista_legajos_activos`
-- (oculta "observaciones", que las entenderíamos como confidenciales)**

```
CREATE OR REPLACE VIEW vista_legajos_activos AS
SELECT
    e.nombre,
    e.apellido,
    l.nroLegajo,
    l.categoría,
    l.estado,
    l.fechaAlta
FROM
    legajo AS l
JOIN
    empleado AS e ON l.empleado_id = e.id
WHERE
    l.estado = 'ACTIVO'
    AND l.eliminado = FALSE
    AND e.eliminado = FALSE;
```

¡Excelente! Tus dos vistas están muy bien diseñadas y cumplen perfectamente el objetivo.

1. En la vista_empleados_publica, aciertas al ocultar columnas como los IDs, dni, email, etc. Es una vista de 'directorio' perfecta.
2. En la vista_legajos_activos, cumples exactamente el requisito: oculta la columna observaciones (que identificaste bien como sensible) y además filtra por estado = 'ACTIVO', lo cual es un filtro de filas muy lógico para un gerente.

El doble chequeo de e.eliminado = FALSE y l.eliminado = FALSE en ambas vistas es una práctica muy robusta. ¡Muy bien hecho!

Conectando las piezas

Ahora viene la parte clave del ejercicio, donde unimos lo que hicimos antes (el usuario) con lo que acabamos de hacer (las vistas).

Pensemos juntos:

Crear la vista es la preparación. ¿Cómo usamos esta vista para implementar la seguridad? Recuerda a nuestro usuario 'tpi_bd'@'localhost'. En el ejercicio anterior, le dimos permisos SELECT sobre las tablas reales (empleado y legajo).

- Pregunta 1: ¿Qué problema de seguridad sigue existiendo si le dejamos el SELECT sobre la tabla empleado? (Incluso si solo tiene SELECT, ¿qué datos "sensibles" sigue pudiendo ver?)
- Pregunta 2: ¿Qué deberíamos hacer para "forzar" a que ese usuario 'tpi_bd' solo pueda ver los datos "limpios" de la vista_empleados_publica y no los datos sensibles de la tabla empleado?

Pista: Involucra dos comandos SQL. Uno para quitar un permiso y otro para dar uno nuevo.



- IX. Si sigue SELECT en la tabla empleado, pese a haber creado una vista que oculte columnas, podría acceder igual a la tabla original, ya que sigue manteniendo privilegios al respecto.

Lo que si nos faltó fue añadir nuevos permisos en base a las vistas:

```
GRANT      SELECT      ON      empleados.vista_empleados_publica      TO
'tpi_bd'@'localhost';
GRANT SELECT ON empleados.vista_legajos_activos TO 'tpi_bd'@'localhost';
Tengo una consulta. Siempre debemos usar FLUSH PRIVILEGES; luego de
cambiar permisos?
```

¡Tu análisis es 100% correcto!

Ese es exactamente el problema. Crear la vista es la preparación, pero no sirve de nada si el usuario puede simplemente "rodearla" y acceder a la tabla original.

Tus comandos GRANT para las vistas también son correctos. Ese es el paso 1: dar el acceso nuevo y limitado.

El paso que te falta

Ahora, para completar la implementación de seguridad, necesitas el paso 2: quitar el acceso antiguo.

¿Cuál es el comando opuesto a GRANT? ¿Cómo le quitarías los permisos (SELECT, INSERT, UPDATE) sobre las tablas empleado y legajo al usuario tpi_bd?

Pista: La sintaxis es muy parecida a GRANT.

SQL

```
REVOKE [permisos] ON [dónde] FROM [quién];
```

Intenta escribir los comandos REVOKE que necesitarías.

Tu consulta sobre FLUSH PRIVILEGES;

¡Excelente pregunta! Es una de las dudas más comunes en MySQL.

La respuesta corta es: No, ya no es necesario (casi nunca).

Explicación: 

Piensa que los comandos como CREATE USER, GRANT y REVOKE son "comandos oficiales" de seguridad. Cuando los ejecutas, MySQL es lo suficientemente inteligente como para saber que los permisos cambiaron y los actualiza en su memoria automáticamente.

¿Cuándo SÍ se usaba FLUSH PRIVILEGES?

Se usaba (y a veces se usa) solo si modificas las tablas de permisos directamente. Por ejemplo, si en lugar de usar CREATE USER, hicieras esto (que no se recomienda):

```
INSERT INTO mysql.user (Host, User, ...) VALUES ...;
```

Como hiciste un INSERT manual, MySQL no se da cuenta solo. Ahí sí necesitarías ejecutar FLUSH PRIVILEGES; para decirle: "¡Oye, relee las tablas de permisos porque las modifiqué a mano!".

En resumen: Para tu trabajo práctico, usando GRANT y REVOKE, nunca necesitas usar FLUSH PRIVILEGES;. Los cambios se aplican al instante.

- X. Claro, pero el asunto es que igual el usuario debe seguir teniendo el privilegio SELECT sobre empleado, ya que sino vulneraríamos CRUD. Además, la consigna no indica que el usuario deje de acceder a las tablas originales, simplemente señala que creemos las vistas.

Es una observación muy aguda, y has tocado el corazón del problema de seguridad. Entiendo tu lógica: la consigna dice "diseñar vistas" y ya lo hicimos. Pero recordemos la meta principal de la Etapa 4: "Implementar medidas de seguridad..." y "...usar vistas para exponer datos".

Pensemos juntos sobre esto:

1. Diseñamos vista_empleados_publica con el objetivo de ocultar el dni y el email.
2. Pero, como bien dices, el usuario tpi_bd todavía tiene SELECT sobre la tabla empleado original.
3. Por lo tanto, el usuario puede simplemente ignorar la vista, ejecutar SELECT dni FROM empleado; y... ver la información sensible. 😱

En ese escenario, ¿hemos "implementado" la seguridad? ¿O solo hemos "diseñado" una vista que el usuario no está obligado a usar?

El "Clic" Conceptual

La idea de "usar vistas para exponer datos" (como dice tu consigna) es que la vista se convierta en la única puerta de entrada para leer los datos.

- Quitas la llave de la "puerta de servicio" (la tabla original).
- Le das al usuario la llave de la "puerta principal" (la vista), que solo muestra el "lobby" (los datos no sensibles).

¿Qué pasa con CRUD?

¡Tienes razón! Si le quitas el SELECT a la tabla empleado, rompes la 'R' (Read) de CRUD sobre esa tabla.

Pero se lo estás reemplazando por un SELECT sobre la vista. El usuario sigue pudiendo leer, solo que ahora lee la "versión limpia".

El esquema de permisos final debería ser así:

- Para tpi_bd:
 - Sobre empleado (la tabla): INSERT, UPDATE. (Para que pueda crear y modificar).
 - Sobre empleado (la tabla): Quitarle SELECT y DELETE (como sugeriste).
 - Sobre vista_empleados_publica (la vista): SELECT. (Para que pueda leer los datos filtrados).

De esta forma, el usuario puede gestionar empleados (C y U), pero solo puede leer (R) la información pública. Has implementado la seguridad perfectamente.

Tu Siguiente Paso

Basado en esto, ¿cómo modificarías tus comandos de permisos?

Necesitarás usar REVOKE para quitar el SELECT de la tabla original y luego GRANT para dar SELECT solo a la vista (como ya escribiste antes).

Intenta escribir los comandos REVOKE específicos para quitar SELECT de empleados.empleado y empleados.legajo.

XI. De momento decidimos no usar REVOKE para esto, ya que queremos evitar conflictos con el resto de las actividades. Continuemos con la próxima actividad.

Entendido. Es una decisión válida para mantener el alcance del trabajo práctico acotado. Sigamos con el próximo punto de la Etapa 4: "...evitar SQL injection mediante consultas parametrizadas..."

Este es uno de los conceptos de seguridad más importantes. A diferencia de GRANT o CREATE VIEW, esto no es un comando que ejecutas en la base de datos, sino que se refiere a cómo escribes el código en tu aplicación (el programa en Java, Python, PHP, etc.) que se conecta a la base de datos.

El Problema: La Concatenación de Strings

Imagina que tu aplicación necesita buscar un empleado por su dni. Un programador novato podría escribir el código (en pseudocódigo) para armar la consulta SQL así:

Java

```
// El DNI viene de un campo de texto que el usuario llenó
String dni_del_usuario = "22.333.444";

// Se "arma" la consulta pegando el texto del usuario
String consulta_sql = "SELECT * FROM empleado WHERE dni = '" + dni_del_usuario + "'.";

// Se ejecuta la consulta...
```

Esto funciona bien si el usuario es honesto y escribe "22.333.444". La consulta final que recibe MySQL es:

```
SELECT * FROM empleado WHERE dni = '22.333.444';
```

Pista: El Ataque

Ahora, ¿qué pasa si un usuario malicioso, en lugar de un DNI, escribe esto en el campo de texto?:

' OR '1'='1

Piénsalo: si tomas ese texto y lo "pegas" (concatenas) en la consulta SQL de arriba...

Pregunta: ¿Cuál es la consulta SQL completa que terminaría recibiendo y ejecutando tu servidor MySQL? ¿Qué crees que haría esa consulta?

XII. No, el siguiente punto es: Ejecutar al menos 2 pruebas de integridad (ej. duplicación de PK, inserción fuera de rango, violación de FK).

Entendido. Esta es una parte clave de la "integridad de la base".

El objetivo aquí es demostrar que las "reglas" (constraints) que definiste en tu CREATE TABLE (tu archivo 01_esquema.sql) están funcionando. Tienes que escribir consultas INSERT o UPDATE que intenten romper esas reglas.

La "prueba" de que la integridad funciona es que MySQL te devuelva un error.

Veamos tu esquema y las 3 pruebas que mencionaste. Tienes todo lo necesario.

Pista 1: Prueba de "Duplicación" (PK o UNIQUE)

Tu tabla empleado tiene esta regla excelente:

dni VARCHAR(15) NOT NULL UNIQUE

Esto significa que no pueden existir dos empleados con el mismo DNI.

- ¿Cómo lo probamos?

1. Primero, inserta un empleado nuevo (con un DNI, digamos '12345678'). Este INSERT debe funcionar.
2. Luego, intenta insertar un segundo empleado, pero usando exactamente el mismo DNI ('12345678').

¿Qué consulta INSERT escribirías para ese segundo paso? ¿Qué error esperas que te devuelva MySQL?

Pista 2: Prueba de "Violación de FK" (Foreign Key)

Tu tabla legajo tiene esta regla:

FOREIGN KEY (empleado_id) REFERENCES empleado(id)

Esto significa que no puedes crear un legajo que apunte a un empleado_id que no exista en la tabla empleado.

- ¿Cómo lo probamos?

- Intenta insertar un nuevo legajo y, en el campo empleado_id, pon un número que sepas que no existe. Por ejemplo, 99999.

¿Cómo escribirías ese INSERT INTO legajo ...? ¿Qué error debería darte?

Pista 3: Prueba de "Fuera de Rango"

Esta es una gran prueba para tu columna legajo.estado:

estado ENUM('ACTIVO', 'INACTIVO') NOT NULL

La regla ENUM obliga a que el valor sea únicamente 'ACTIVO' o 'INACTIVO'.

- ¿Cómo lo probamos?

- Intenta insertar un nuevo legajo pero, en la columna estado, pon un valor que no esté en la lista, por ejemplo, 'EN_LICENCIA'.

Eige dos de estas tres pruebas e intenta escribir las consultas SQL que deberían fallar. Esos fallos serán tu respuesta para el trabajo práctico.

XIII. En base a tus pistas 1 y 2 construimos dos querys con INSERT INTO para rerealizar las pruebas de integridad correspondientes:

-- Pruebas de Integridad

-- Prueba Nº1: UNIQUE

-- El valor en `dni` debe preexistir en la base de datos / tabla, sino debe ser insertado.

**INSERT INTO empleado (nombre, apellido, dni, email, fechalngreso, area)
VALUES**

```
('Juan', 'Rodríguez', '250265541', 'juan.rodriguez@dominio.com', '2024-05-10',  
'Recursos Humanos');  
-- Prueba Nº2: Violación de la FK en `legajo`  
-- Se ingresa un valor que no existe previamente y que al agregarlo solo en  
`legajo` lanza error, ya que primero debe existir en `empleado`  
INSERT INTO legajo (nroLegajo, categoria, estado, fechaAlta, observaciones,  
empleado_id)  
VALUES (  
    'LEG-999999',  
    'Junior',  
    'ACTIVO',  
    '2021-06-19',  
    'Prueba de FK: Empleado no existe',  
    999999           -- ID de empleado que no existe, que corresponde al  
valor asignado para `empleado_id`  
);  
Considerás que es correcto? pudimos realizar las pruebas correctamente
```

¡Buen trabajo! Vamos a revisar las dos pruebas.

Prueba Nº2: Violación de la FK (Foreign Key)

Esta prueba es perfecta. 🎉

Tu lógica es impecable:

1. Estás intentando insertar un legajo (LEG-999999).
2. Le asignas un empleado_id (999999) que (con casi total seguridad) no existe en la tabla empleado.
3. Tu regla FOREIGN KEY (empleado_id) REFERENCES empleado(id) debe activarse.

Resultado esperado: La consulta debe fallar. MySQL te dará un error claro, algo como Cannot add or update a child row: a foreign key constraint fails.

Ese error es tu demostración exitosa. Has probado que la base de datos mantiene su integridad. ¡Genial!

Prueba Nº1: UNIQUE (Restricción Única)

Aquí hay un pequeño detalle conceptual. Miremos tu consulta:

SQL

```
INSERT INTO empleado (nombre, apellido, dni, email, fechalIngreso, area)
```

```
VALUES
```

```
('Juan', 'Rodríguez', '250265541', 'juan.rodriguez@dominio.com', '2024-05-10', 'Recursos Humanos');
```

Pensemos juntos: Si el DNI 250265541 no existe todavía en tu tabla empleado, ¿qué pasará cuando ejecutes esta consulta?

... Probablemente tendrá éxito y Juan Rodríguez será insertado.

Pero el objetivo de la prueba de integridad es demostrar que la base de datos te detiene cuando intentas romper una regla. La regla UNIQUE te prohíbe insertar duplicados. Para probar que la regla funciona, necesitas intentar insertar un DNI que ya exista.

Pista (Cómo ajustar la Prueba 1)

Tu consulta actual es el Paso A (la "preparación").

1. Paso A (Preparación): Ejecutas ese INSERT de Juan Rodríguez. (Funciona).
2. Paso B (La Prueba Real): ¿Qué consulta INSERT deberías escribir inmediatamente después para intentar insertar, por ejemplo, a 'Laura Gómez', pero usando intencionalmente el mismo DNI que Juan ('250265541')?

Ese segundo INSERT es el que debe fallar (con un error de Duplicate entry ... for key 'dni'). Ese fallo será tu prueba de que la restricción UNIQUE funciona.

Resumen: Tu Prueba 2 es perfecta. Tu Prueba 1 está a un solo paso de serlo: solo necesitas agregar la consulta que intenta crear el duplicado.

XIV. Sí, pero el dni existe en la prueba, justamente a sabiendas de eso lo ejecutamos así. En caso contrario hubiésemos creado otra consulta

¡Ah, perfecto! Entendido.

Disculpa la confusión. Si ejecutas esa consulta (Prueba N°1) sabiendo que el DNI '250265541' ya existe, entonces tu prueba es absolutamente correcta.

Mi error fue asumir que era la primera vez que se insertaba.

En ese caso, tu plan es impecable:

- Prueba N°1 (UNIQUE): Intentas insertar un empleado con un DNI que ya existe. MySQL debe fallar y devolver un error Duplicate entry. ✓
- Prueba N°2 (Foreign Key): Intentas insertar un legajo con un empleado_id que no existe. MySQL debe fallar y devolver un error de a foreign key constraint fails. ✓

Ambas pruebas son exitosas porque demuestran que las restricciones de integridad de tu base de datos están activas y funcionan, impidiendo la entrada de datos "malos".

¡Excelente trabajo con esta actividad!

¿Cuál es el siguiente y último punto de la Etapa 4? (Creo que mencionaste algo sobre SQL Injection).

El último punto es el siguiente:

Implementar en Java al menos 1 consulta parametrizada con PreparedStatement para prevenir SQL Injection o SQL (procedimiento almacenado sin SQL dinámico), con prueba anti-inyección documentada.

Lo que sucede aquí es que no comenzamos con la implementación en Java. Si o sí debemos usar Java para esto?

¡Esa es una pregunta clave! Y la respuesta está en la propia consigna que me transcribiste. Revisémosla con atención:

"Implementar en Java al menos 1 consulta parametrizada con PreparedStatement... o SQL (procedimiento almacenado sin SQL dinámico), con prueba anti-inyección documentada."

La palabra clave es "o".

La consigna te da dos caminos válidos para cumplir este punto. Solo necesitas elegir uno:

1. Camino A: El de Java (PreparedStatement).
2. Camino B: El de SQL (procedimiento almacenado).

Dado que me dices que no han comenzado con la implementación en Java, el Camino B parece ser el indicado para tu trabajo práctico.

Podemos demostrar exactamente el mismo principio de seguridad (prevenir SQL Injection) usando solamente un Procedimiento Almacenado (Stored Procedure) en MySQL.

¿Te parece que exploremos ese camino? Podemos empezar por definir qué es un procedimiento almacenado y cómo nos protege.

XV. Bien, entonces continuemos con el camino B

Perfecto. El Camino B (Procedimiento Almacenado) es una solución excelente y 100% SQL.

¿Cómo nos protege un Procedimiento Almacenado?

El truco de la "Inyección SQL" (SQL Injection) funciona cuando la base de datos confunde los datos del usuario con un comando SQL.

- Consulta Vulnerable (con concatenación):
"SELECT * FROM tabla WHERE dni = " + input_del_usuario + ";"
Si el usuario escribe ' OR '1'='1, la consulta se convierte en:
SELECT * FROM tabla WHERE dni = " OR '1'='1';
...y la base de datos obedece al comando OR '1'='1'.
- Procedimiento Almacenado (Consulta Parametrizada):
Un procedimiento almacenado nos obliga a separar la "plantilla" de la consulta de los "datos".
 1. Tú defines la plantilla: SELECT * FROM tabla WHERE dni = ?;
 2. Luego, le pasas los datos: (' OR '1'='1')
 3. MySQL nunca mezcla ambos. Recibe la plantilla y "mete" el dato del usuario en el espacio ?. Trata el input del usuario siempre como texto, nunca como un comando.
 4. Buscará un empleado cuyo DNI sea, literalmente, la cadena de texto ' OR '1'='1'. No lo encontrará, y la consulta no devolverá nada. ¡El ataque falló!

Pista 1: El Escenario

Vamos a crear un procedimiento almacenado que busque un empleado por su dni. Lo llamaremos sp_buscar_empleado.

Este procedimiento necesitará recibir un dato: el DNI que queremos buscar. A esto se le llama un parámetro de entrada (un parámetro IN).

Pista 2: La Sintaxis

La sintaxis para crear un procedimiento en MySQL se ve así. Presta atención a los DELIMITER, son importantes.

SQL

```
-- Cambiamos el delimitador de ";" a "://" para que MySQL  
-- no se confunda y piense que el procedimiento termina en el primer ";".  
DELIMITER //
```

```
CREATE PROCEDURE sp_buscar_empleado(IN dni_param VARCHAR(15))  
BEGIN  
    -- Aquí va tu consulta SQL "limpia".  
    -- Usa el parámetro (dni_param) en lugar de un valor 'quemado'.  
  
    SELECT * FROM empleado  
    WHERE dni = dni_param;  
  
END //
```

```
-- Devolvemos el delimitador a la normalidad  
DELIMITER ;
```

Analicemos esto:

- CREATE PROCEDURE sp_buscar_empleado(...): Le da un nombre.
- IN dni_param VARCHAR(15): Esta es la parte clave. Creamos un "lugar" llamado dni_param que recibirá el texto del DNI. Al definirlo como VARCHAR(15), le decimos a MySQL que "lo que sea que entre aquí, trátalo como un simple texto de hasta 15 caracteres".
- WHERE dni = dni_param;: Usamos el parámetro. La consulta es fija, no se puede "romper" desde afuera.

Tu Siguiente Paso

Intenta copiar (o reescribir) ese bloque de código en tu script SQL para crear el procedimiento.

Una vez que lo hayas creado, el siguiente paso será la "prueba anti-inyección documentada". ¿Cómo se te ocurre que podríamos "llamar" a este procedimiento y pasarle un valor normal, y luego un valor malicioso, para probar que funciona?

Podríamos llamarlo usando CALL y el nombre del procedimiento que mencionaste, cierto?

¡Exacto! Tienes toda la razón. 

El comando para ejecutar un procedimiento almacenado en MySQL es, de hecho, CALL.

La sintaxis es:

SQL

```
CALL nombre_del_procedimiento(lista_de_argumentos);
```

La Prueba Anti-Inyección (Tu Siguiente Paso)

Ahora, para cumplir con la consigna ("con prueba anti-inyección documentada"), necesitas demostrar qué pasa en dos escenarios.

Supongamos que en tu tabla empleado Sí existe alguien con el DNI '250265541' (el que usamos en la prueba anterior).

Prueba A: La consulta "honesta"

Vas a llamar al procedimiento pasándole un DNI normal:

SQL

```
CALL sp_buscar_empleado('250265541');
```

Prueba B: El intento de "Inyección SQL"

Ahora, vas a llamar al mismo procedimiento, pero le pasarás la cadena de texto maliciosa que vimos antes:

SQL

```
CALL sp_buscar_empleado(" OR '1='1");
```

Tu Tarea:

Piensa y descríbeme: ¿Cuál es el resultado que esperas obtener de la Prueba A y cuál es el resultado que esperas de la Prueba B?

Explicar por qué la Prueba B no te devuelve todos los empleados (como lo haría una consulta vulnerable) es el corazón de esta actividad.

No se necesita delimitador para crear el PROCEDURE:

-- SQL - Procedimiento almacenado - Evitando la SQL Injection

-- No se necesita delimitador

```
CREATE PROCEDURE sp_buscar_empleado(IN dni_param VARCHAR(15))
BEGIN
    SELECT * FROM empleado
    WHERE dni = dni_param
    AND eliminado = FALSE;
END;
```

Luego de unas pruebas en Dbeaver pude darme cuenta. Podrías aclararme más al respecto?

¡Ah, qué gran observación! Tienes toda la razón y es un detalle técnico muy importante. Mi disculpa por la confusión. Tienes razón en que DBeaver (y otros clientes gráficos modernos como MySQL Workbench) son más inteligentes que la consola simple de MySQL. Déjame aclararte por qué no lo necesitaste:

Por qué SÍ se usa DELIMITER

El comando DELIMITER no es un comando de SQL. No es para el servidor MySQL. Es un comando exclusivo para el cliente de línea de comandos (mysql.exe o el terminal mysql).

En la consola simple, el cliente lee tu texto y, en cuanto ve un punto y coma (;), corta el comando ahí y lo envía al servidor.

Si intentas pegar esto en la consola:

SQL

```
CREATE PROCEDURE sp_buscar_empleado(IN dni_param VARCHAR(15))
BEGIN
    SELECT * FROM empleado
    WHERE dni = dni_param; <-- ¡PROBLEMA AQUÍ!
END;
```

La consola vería el primer ; y enviaría CREATE PROCEDURE ... WHERE dni = dni_param; al servidor. El servidor daría un error de sintaxis porque falta el END.

Por eso, en la consola, le dices DELIMITER // para que "ignore" los ; y solo envíe el bloque completo cuando vea un //.

Por qué NO lo necesitaste en DBeaver

DBeaver no es una consola simple. Tiene un analizador de SQL (un parser).

Cuando ejecutas un script (usualmente con F5 o el botón "Ejecutar Script", no el de "Ejecutar Sentencia"), DBeaver lee todo tu texto. Identifica que estás en un bloque CREATE PROCEDURE ... BEGIN ... END;.

Sabe que los ; dentro del BEGIN...END son parte del cuerpo del procedimiento, no el final de la instrucción completa. Por lo tanto, DBeaver envía de forma inteligente todo el bloque (CREATE PROCEDURE ... END;) al servidor MySQL de una sola vez.

En resumen: tú estás en lo correcto. En un entorno gráfico moderno como DBeaver, el DELIMITER es (casi siempre) innecesario porque la herramienta es lo suficientemente lista para entender la estructura de tu código. Mi pista era para el cliente de consola estándar.

Tu Procedimiento

Por cierto, tu versión mejorada del procedimiento es excelente.

SQL

```
CREATE PROCEDURE sp_buscar_empleado(IN dni_param VARCHAR(15))
BEGIN
    SELECT * FROM empleado
    WHERE dni = dni_param
    AND eliminado = FALSE;
END;
```

Agregar AND eliminado = FALSE; es una práctica de negocio muy buena, ya que te aseguras de que el buscador solo devuelva empleados activos.

De vuelta a la Prueba de Seguridad

Ahora que el procedimiento está creado, volvamos a la demostración de seguridad. Vas a ejecutar estas dos pruebas:

1. Prueba A (Honesto): CALL sp_buscar_empleado('250265541'); (Suponiendo que ese DNI sí existe y no está eliminado).
2. Prueba B (Ataque): CALL sp_buscar_empleado("") OR '1'='1";

Para completar tu "prueba anti-inyección documentada", describe:

- ¿Qué resultado esperas de la Prueba A?
- ¿Qué resultado esperas de la Prueba B? (¿Y por qué no te devuelve todos los empleados?)

Podrías volver a explicarme qué es lo que hace exactamente esta "plantilla"?

¡Claro que sí! Piénsalo de esta manera:

La "plantilla" (ya sea en un PreparedStatement de Java o en tu CREATE PROCEDURE de SQL) es como un contrato blindado que haces con la base de datos.

Este "contrato" (la plantilla) tiene la estructura de la consulta, que es fija e inalterable.

La Consulta Vulnerable (Sin Plantilla)

Imagina que le das a MySQL tus instrucciones en una hoja de papel borrador.

- Tú escribes: SELECT * FROM empleado WHERE dni = '
- Luego, tomas lo que escribió el usuario (el "dato") y lo pegas al final.
- Si el usuario escribió ' OR '1'='1, tu hoja ahora dice:
SELECT * FROM empleado WHERE dni = " OR '1'='1"
- Le entregas esta hoja a MySQL. MySQL la lee y dice: "Ah, perfecto. Quieres que busque un DNI vacío O donde 1 sea igual a 1... ¡OK, te devuelvo todo!".

El usuario cambió la estructura de tus instrucciones. ☺

La Consulta Segura (Con Plantilla)

Ahora, imagina que usas la "plantilla" (tu sp_buscar_empleado).

1. Paso 1: Entregas el "Contrato" (La Plantilla)
Tú le das a MySQL tu CREATE PROCEDURE. MySQL lo lee, lo analiza y lo "compila". Dice: "Entendido. He guardado esta plantilla llamada sp_buscar_empleado. Siempre buscará en la tabla empleado, comparará la columna dni y la columna eliminado. Y espera un solo dato (dni_param) para poner en ese hueco".
La estructura SELECT ... WHERE ... AND ... queda grabada en piedra.
2. Paso 2: Entregas los "Datos" (La Llamada)
Cuando ejecutas CALL sp_buscar_empleado("") OR '1'='1";, no estás enviando un comando SQL.
Solo le estás diciendo a MySQL: "Oye, ¿recuerdas esa plantilla sp_buscar_empleado? Bueno, aquí tienes el dato que va en el hueco dni_param: es el texto "" OR '1'='1"".
3. Paso 3: MySQL Ejecuta (¡De forma segura!)
MySQL jamás vuelve a armar la consulta. Simplemente toma el texto que le diste (tal cual) y lo pone en el hueco. La consulta que internamente ejecuta es, conceptualmente, esta:
SELECT * FROM empleado WHERE dni = "" OR '1'='1" AND eliminado = FALSE;

MySQL buscará un empleado cuyo DNI sea, literalmente, la cadena de texto "" OR '1'='1'. Como no existe ningún empleado con ese DNI tan raro, la consulta no devuelve nada.

El ataque falló. La plantilla impidió que el dato del usuario se convirtiera en un comando.

Podrías volver a explicarme por qué ' OR '1'='1'

Claro. Es la "llave maestra"  de los ataques de Inyección SQL, y es brillante por lo simple que es.

El objetivo del atacante es escribir un texto que, al "pegarse" en tu consulta, cambie la lógica del comando WHERE para que siempre sea verdadero.

Vamos a analizarlo parte por parte, usando la consulta vulnerable (la que no usa un procedimiento almacenado).

Anatomía del Ataque

Imagina que el código de la aplicación (Java, PHP, etc.) tiene esta línea vulnerable que "arma" el SQL:

Java

```
// VULNERABLE: Se "pega" (concatena) el texto del usuario
String consulta_sql = "SELECT * FROM empleado WHERE dni = " + input_del_usuario +
"';";
```

El programador espera que el usuario solo escriba un DNI.

1. El Ataque: El "Input" Malicioso

El atacante no escribe un DNI. En el campo de texto, escribe esto:
' OR '1'='1

2. La Concatenación: El "Armado" de la Consulta

Ahora, mira lo que pasa cuando Java "pega" ese texto en la consulta_sql:

- Parte 1 (del código): SELECT * FROM empleado WHERE dni = '
- Parte 2 (del atacante): ' OR '1'='1'
- Parte 3 (del código): ';

La consulta final que recibe MySQL es:

```
SELECT * FROM empleado WHERE dni = '' OR '1'='1';
```

3. El Análisis de MySQL: La "Trampa"

MySQL obedece y analiza esa consulta. Mira el WHERE:
WHERE dni = '' OR '1'='1'

MySQL ve una operación OR. Para que el WHERE sea verdadero, solo necesita que una de las dos condiciones sea verdadera.

- Condición 1: dni = "" (¿El DNI es una cadena vacía?)

- Probablemente FALSO para todas las filas.
- Condición 2: '1' = '1' (¿Es el texto '1' igual al texto '1')?
 - Esto es siempre VERDADERO.

El resultado lógico del WHERE es:

WHERE (FALSO) OR (VERDADERO)

... lo cual se evalúa siempre a:

WHERE VERDADERO

La consulta se ha convertido en: SELECT * FROM empleado WHERE VERDADERO;

MySQL interpreta esto como: "Selecciona todas las columnas, de todos los empleados, sin ninguna condición".

Resultado: El atacante acaba de "saltarse" la seguridad del WHERE y ha obtenido toda la tabla de empleados.

Resumen: ¿Por Qué ' OR '1'='1?

- La primera comilla simple (') cierra la comilla que el programador abrió (... WHERE dni = '...').
- OR añade una nueva condición lógica.
- '1'='1' crea una condición que es universal e innegablemente VERDADERA.

Es un texto diseñado para "escaparse" de ser un dato y convertirse en parte del comando.

Tu Procedimiento Almacenado  (la plantilla) evita esto porque trata la cadena entera "" OR '1'='1" como un solo dato para buscar, nunca como un comando.

Etapa 5 – Conurrencia y Transacciones

- I. Intentamos ejecutar dos scripts con dos sesiones iniciadas en SQL y salió este error: **SQL Error [1205] [40001]: Lock wait timeout exceeded; try restarting transaction.**

Claro, analicemos ese error. Es uno de los más comunes cuando se trabaja con transacciones y es fundamental entender por qué ocurre.

Pensemos en una analogía simple: el baño de un tren. 

* **Sesión 1** es una persona que entra al baño y **cierra la puerta con llave** (pone un "lock" o bloqueo).

* **Sesión 2** es otra persona que llega y quiere usar el mismo baño. Se encuentra con la puerta cerrada y decide esperar.

* El "Lock wait timeout" es el tiempo máximo que la segunda persona está dispuesta a esperar. Si la primera persona no sale del baño (libera el bloqueo) en, digamos, 50 segundos, la segunda persona se cansa, se va y dice: "¡No pude entrar!".

Eso es exactamente lo que pasó entre tus dos sesiones de SQL.

¿Qué Sucedió a Nivel Técnico?

1. **Inicio de la Transacción (Sesión 1):** Tu primer script probablemente inició una transacción (quizás con un `START TRANSACTION;`) y luego ejecutó una sentencia que modifica datos, como un `UPDATE`, `DELETE` o `INSERT`. Al hacerlo, el motor de la base de datos (seguramente InnoDB) **bloqueó las filas (o la tabla)** que estaban siendo modificadas^{**}. Este bloqueo es necesario para proteger la integridad de los datos y evitar que dos personas modifiquen lo mismo a la vez.
2. **Intento de Acceso (Sesión 2):** Tu segundo script, en una sesión diferente, intentó leer o modificar **exactamente las mismas filas** que la Sesión 1 había bloqueado.
3. **La Espera:** Como las filas estaban bloqueadas, la base de datos le dijo a la Sesión 2: "Espera un momento, este recurso está ocupado. Te pongo en la cola".
4. **El Timeout:** La Sesión 1 **nunca terminó su transacción** (nunca hizo `COMMIT` para confirmar los cambios o `ROLLBACK` para cancelarlos). Por lo tanto, nunca liberó el bloqueo (nunca "salió del baño"). El tiempo de espera de la Sesión 2 se agotó y la base de datos la canceló para evitar que se quedara esperando indefinidamente, arrojando el error `Lock wait timeout exceeded`.

¿Cómo lo Solucionas y Evitas?

El problema no está en el script de la Sesión 2 (la que dio el error), sino en la **Sesión 1**, que se quedó "colgada" con el bloqueo activo.

* **Solución Inmediata:** Ve a la ventana de la **Sesión 1** y termina la transacción.

- * Si quieres guardar los cambios que hizo: `COMMIT;`
- * Si quieres descartar los cambios: `ROLLBACK;`

Apenas ejecutes uno de esos comandos, el bloqueo se liberará. Después, puedes volver a ejecutar el script en la Sesión 2 y funcionará sin problemas.

* **Buena Práctica para Evitarlo:** Las transacciones deben ser lo más **cortas y rápidas** posible. El patrón correcto es:

1. Iniciar la transacción.
2. Hacer las operaciones de modificación necesarias.
3. Confirmar (`COMMIT`) o revertir (`ROLLBACK`) inmediatamente.

El error suele ocurrir cuando un desarrollador deja una transacción abierta mientras hace otras cosas o se olvida de incluir el `COMMIT` al final del script.

Intenta encontrar en tu primer script dónde quedó la transacción abierta. Si la cierras, el problema se resolverá. ¡Contame si lo encuentras!

II. ¿Cómo realizar un procedimiento almacenado en SQL?

Claro que sí. Un procedimiento almacenado (o *stored procedure*) es una de las herramientas más útiles en SQL.

Piénsalo como una **receta de cocina guardada directamente en la base de datos** . En lugar de escribir los mismos pasos (las mismas sentencias SQL) una y otra vez, creas una receta, le das un nombre y luego simplemente la "llamas" por su nombre para que se ejecute.

¿Por Qué Usarlos?

- * **Reutilización:** Escribes el código una vez y lo usas las veces que quieras.
- * **Simplicidad:** Ocultas lógica compleja detrás de un comando simple. En lugar de escribir 10 líneas de `JOIN`s y `WHERE`'s, solo haces `CALL mi_reporte_complejo();` .
- * **Seguridad:** Puedes dar permiso a un usuario para que ejecute el procedimiento, pero no para que toque las tablas directamente.
- * **Rendimiento:** La base de datos "pre-compila" y optimiza el plan de ejecución del procedimiento, por lo que a menudo es más rápido.

¿Cómo se Crea Uno? (Sintaxis Básica)

La estructura fundamental para crear un procedimiento es la siguiente. ¡Vamos por partes!

1. Cambiar el Delimitador:

Normalmente, SQL termina un comando cuando ve un punto y coma (`;`). Pero nuestro procedimiento tendrá varios punto y coma *dentro* de él. Para evitar que se corte a la mitad, primero le decimos a SQL que use otro símbolo como finalizador, por ejemplo `//` o `\$\$`.

```
```sql
DELIMITER //
````
```

2. La Estructura `CREATE PROCEDURE`:

Aquí es donde definimos nuestra "receta".

```
```sql
CREATE PROCEDURE nombre_del_procedimiento()
````
```

BEGIN

-- Aquí adentro van todas las sentencias SQL que quieras ejecutar.
-- Por ejemplo, un SELECT, un INSERT, un UPDATE, etc.

END //

...

3. Volver al Delimitador Original:

Una vez creado el procedimiento, le decimos a SQL que vuelva a usar el punto y coma.

```
```sql
DELIMITER ;
````
```

Ejemplo 1: Un Procedimiento Simple

Imaginemos que tenemos una tabla `usuarios` y queremos un procedimiento para mostrar todos los usuarios activos.

```
```sql
-- 1. Cambiamos el delimitador
DELIMITER //

-- 2. Creamos el procedimiento
CREATE PROCEDURE sp_mostrar_usuarios_activos()
BEGIN
 -- Esta es la "receta": seleccionar usuarios donde el estado es 'activo'
 SELECT id, nombre, email FROM usuarios WHERE estado = 'activo';
END //

-- 3. Restauramos el delimitador
DELIMITER ;
````
```

¿Y cómo lo usamos? ¡Muy fácil! Con el comando `CALL`:

```
```sql
CALL sp_mostrar_usuarios_activos();
````
```

Al ejecutar esa línea, la base de datos correrá el `SELECT` que guardamos adentro.

Ejemplo 2: Agregando Parámetros (Haciéndolo más útil)

Lo verdaderamente poderoso es poder pasárselo "ingredientes" a nuestra receta. A estos se les llama **parámetros**. El más común es el de entrada (`IN`).

Vamos a crear un procedimiento para buscar un usuario por su `id`.

```
```sql
DELIMITER //

-- Le decimos que aceptará un parámetro de entrada llamado 'id_usuario' de tipo INT
CREATE PROCEDURE sp_buscar_usuario_por_id(IN id_usuario INT)
BEGIN
 -- Usamos el parámetro en la consulta
 SELECT id, nombre, email, estado FROM usuarios WHERE id = id_usuario;
END //

DELIMITER ;
````
```

Para usarlo, simplemente le pasamos el `id` que queremos buscar dentro de los paréntesis:

```
```sql
-- Esto buscará al usuario con id = 5
CALL sp_buscar_usuario_por_id(5);

-- Y esto buscará al usuario con id = 12
CALL sp_buscar_usuario_por_id(12);
````
```

Ahora te toca a vos. Intenta crear un procedimiento simple para tu proyecto. Por ejemplo, uno que inserte un nuevo registro en alguna de tus tablas. ¡Si te trabas, aquí estoy para ayudarte!

III. ¿Cómo conviene hacer un logging de errores en SQL?

Excelente pregunta. Hacer logging de errores directamente en SQL es una técnica muy potente, especialmente para capturar problemas que ocurren dentro de **procedimientos almacenados, funciones o triggers**, lógica que vive y se ejecuta 100% dentro de la base de datos.

Piénsalo como instalar un **cuaderno de bitácora dentro del propio motor del barco** . Si algo falla en la sala de máquinas (la base de datos), queda registrado ahí mismo, sin necesidad de que el capitán (la aplicación) se dé cuenta en ese instante.

El método más conveniente y robusto se basa en dos componentes principales:

1. Una **tabla dedicada** para guardar los logs.
2. Un **manejador de errores (`HANDLER`)** dentro de tu código SQL para atrapar el error y escribir en esa tabla.

Paso 1: Crear la Tabla de Logs

Primero, necesitas un lugar para almacenar los errores. Una tabla de logs bien diseñada es la base de todo.

```
```sql
CREATE TABLE `tabla_logs` (
 `id_log` INT AUTO_INCREMENT PRIMARY KEY,
 `fecha_hora` TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
 `nivel` VARCHAR(10) NOT NULL, -- 'ERROR', 'WARNING', 'INFO'
 `nombre_procedimiento` VARCHAR(100), -- ¿Dónde ocurrió?
 `codigo_error` INT, -- El código de error de SQL (ej. 1062 para clave duplicada)
 `mensaje_error` TEXT, -- El mensaje que devuelve la base de datos
 `contexto` JSON -- Un campo flexible para guardar datos útiles (ej. los parámetros de
entrada)
);
```

```

* **`contexto` como JSON:** Usar un campo `JSON` es una práctica moderna y muy recomendada, ya que te permite guardar un objeto con los valores de los parámetros que causaron el error, dándote una "foto" exacta de la situación.

Paso 2: Usar un `HANDLER` para Capturar Errores

Dentro de un procedimiento almacenado, no puedes usar un `try-catch` como en otros lenguajes. En su lugar, MySQL/MariaDB nos ofrece el `HANDLER`. Es una directiva que le dice a la base de datos: "Si durante la ejecución de este bloque ocurre un error de este tipo, en lugar de detenerte y fallar, ejecuta este otro bloque de código".

La sintaxis clave es: `DECLARE HANDLER FOR SQLEXCEPTION ...`

Ejemplo Completo: Procedimiento que Loguea sus Propios Errores

Vamos a crear un procedimiento para insertar un nuevo `producto`. La tabla `productos` tiene una restricción `UNIQUE` en la columna `codigo_sku`. Si intentamos insertar un SKU que ya existe, se producirá un error (código `1062`). Vamos a capturarlo y registrarlo. 

```
```sql
DELIMITER //

CREATE PROCEDURE sp_insertar_producto(
 IN p_nombre VARCHAR(100),
 IN p_codigo_sku VARCHAR(50),
 IN p_precio DECIMAL(10, 2)
)
BEGIN
 -- Declaramos un manejador para cualquier error de SQL (SQLEXCEPTION)
 DECLARE EXIT HANDLER FOR SQLEXCEPTION
 BEGIN
 -- Cuando un error ocurre, este bloque se ejecuta
 DECLARE v_error_code INT;
 DECLARE v_error_message TEXT;

 -- Obtenemos la información del error que acaba de ocurrir
 GET DIAGNOSTICS CONDITION 1
 v_error_code = MYSQL_ERRNO,
 v_error_message = MESSAGE_TEXT;

 -- Insertamos el error en nuestra tabla de logs
 INSERT INTO tabla_logs (nivel, nombre_procedimiento, codigo_error, mensaje_error,
 contexto)
 VALUES (
 'ERROR',
 'sp_insertar_producto',
 v_error_code,
 v_error_message,
 JSON_OBJECT(
 'nombre', p_nombre,
 'codigo_sku', p_codigo_sku,
 'precio', p_precio
)
);
);

 -- Opcional: Podrías querer que el procedimiento le devuelva una señal de error a la
 aplicación
 -- SELECT 'Ocurrió un error, fue registrado' AS resultado;
END;

-- Este es el código "normal" del procedimiento.
-- Si esta línea falla, el HANDLER de arriba se activará.
INSERT INTO productos (nombre, codigo_sku, precio)
VALUES (p_nombre, p_codigo_sku, p_precio);
```

```
-- Si todo va bien, también podrías registrar un log de 'INFO'
-- INSERT INTO tabla_logs (nivel, nombre_procedimiento, mensaje_error) VALUES
('INFO', 'sp_insertar_producto', 'Producto insertado con éxito');
```

END //

DELIMITER ;  
...

### ### Prueba de Fuego ✓

1. \*\*Ejecución exitosa:\*\*

```
'CALL sp_insertar_producto('Laptop Modelo Z', 'SKU-LAP-001', 1500.00);'
(Esto debería funcionar y no registrará ningún error).
```

2. \*\*Ejecución que causa un error:\*\*

```
'CALL sp_insertar_producto('Otro Producto', 'SKU-LAP-001', 1200.00);'
*(Esto fallará porque el SKU ya existe. El procedimiento no se detendrá bruscamente,
sino que el `HANDLER` se activará).*
```

3. \*\*Revisar la bitácora:\*\*

```
'SELECT * FROM tabla_logs;'
```

Verás una nueva fila en tu tabla `tabla\_logs` con todos los detalles del error de clave duplicada, incluyendo el `codigo\_error` 1062 y el `contexto` con los datos que intentaste insertar.

Este método es la forma más limpia y profesional de manejar errores dentro de SQL. Te da un control total y un registro invaluable para depurar problemas.

## IV. ¿Cómo es la lógica de un bucle de reintentos con un handler específico de error y un backoff?

La lógica de un bucle de reintentos se vuelve más precisa y eficiente cuando se usa un \*\*handler específico de error\*\* y una estrategia de \*\*backoff\*\*. Este enfoque es una optimización del patrón general, ya que te permite reaccionar de manera diferente a cada tipo de error sin necesidad de anidar complejas sentencias `IF` dentro de un manejador genérico.

Pensemos en ello como tener \*\*dos líneas de atención telefónica especializadas\*\* en lugar de una sola línea general. ☎

\* \*\*Línea 1 (Handler para Deadlock):\*\* Su único trabajo es detectar la señal de "ocupado" (`deadlock`). Cuando la escucha, no necesita analizar nada más; simplemente cuelga y anota "reintentar".

\* \*\*Línea 2 (Handler General):\*\* Atiende cualquier otra llamada problemática (un número que no existe, una línea cortada, etc.). Su trabajo es tomar nota detallada del problema y dar por terminado el intento.

Este sistema es más limpio porque cada `HANDLER` tiene una única responsabilidad.

-----

### ### Los Componentes Clave de la Lógica Mejorada

#### 1. \*\*Handlers Específicos (`DECLARE HANDLER FOR error\_code`):\*\*

En lugar de usar un `HANDLER FOR SQLEXCEPTION` para todo, declaras manejadores para los códigos de error específicos que quieras tratar de forma especial. El más importante para reintentos es el \*\*código `1213` para deadlock\*\*.

\* \*\*Handler para Deadlock:\*\* `DECLARE HANDLER FOR 1213 BEGIN ... END;`

\* Este bloque de código se ejecutará \*\*únicamente\*\* si ocurre un deadlock. Su única tarea es activar una bandera o variable (ej. `SET v\_is\_deadlock = TRUE;`) para que el bucle principal sepa que debe reintentar.

\* \*\*Handler General de Respaldo:\*\* `DECLARE HANDLER FOR SQLEXCEPTION BEGIN ... END;`

\* Este manejador atrapará \*\*cualquier otro error\*\* que no sea un deadlock. Su tarea es registrar el error como fatal y activar otra bandera para indicarle al bucle que debe detenerse por completo.

#### 2. \*\*Mecanismo de Backoff (`SLEEP()`):\*\*

El "backoff" es simplemente la \*\*pausa estratégica\*\* que realizas antes de reintentar. Es crucial para no volver a causar el mismo problema de inmediato. Si dos transacciones chocaron, darles una pausa permite que una de ellas termine y libere los recursos, despejando el camino para el reintento.

\* \*\*¿Por qué es importante?\*\* Intentar de nuevo instantáneamente tiene una alta probabilidad de recrear el mismo deadlock. La pausa rompe el ciclo de contención.

\* \*\*Implementación:\*\* Se usa la función `SLEEP(segundos)`. Una buena práctica es introducir un pequeño factor aleatorio (`RAND()`) para evitar que dos procesos que fallaron al mismo tiempo reintenten exactamente en el mismo instante. Por ejemplo: `DO SLEEP(0.5 + RAND());` hará una pausa de entre 0.5 y 1.5 segundos.

Este enfoque es más robusto y legible. El código dentro de cada manejador es mínimo, y la lógica principal del bucle se encarga de orquestar las acciones (reintentar con pausa o abortar) basándose en las banderas que los manejadores activaron.

## ANEXO II: Links a repositorio GitHub y Video

Repositorio: [https://github.com/BrunoGVapore/BD1\\_TFI](https://github.com/BrunoGVapore/BD1_TFI)

Video: <https://youtu.be/JVb5A0feVII>