

# TECNICATURA UNIVERSITARIA EN PROGRAMACIÓN A DISTANCIA

## TRABAJO INTEGRADOR

### Bases de datos

#### Integrantes:

Hugo Catalan

Matias Carro

Ignacio Carné

Gabriel Carbajal

**Materia:** Bases de Datos

**Comision 11**

**Fecha de entrega:** 23/10/2025

**Link al repositorio del trabajo (Con los Scripts SQL):**

[https://github.com/MatiasManuelCarro/UTNTUPad\\_BasesDeDatos1\\_TPI/tree/main/Scripts](https://github.com/MatiasManuelCarro/UTNTUPad_BasesDeDatos1_TPI/tree/main/Scripts)

**Link al video del trabajo:**

<https://www.youtube.com/watch?v=A9338Lvp0NA>

<b>Integrantes:</b> .....	<b>1</b>
<b>Resumen ejecutivo</b> .....	<b>4</b>
<b>Reglas de negocio - Gestión de Usuarios y Credenciales</b> .....	<b>4</b>
<b>Modelo Relacional y Constraints</b> .....	<b>5</b>
Tabla: usuario.....	5
Tabla: credencial_acceso.....	7
<b>Creación de Tablas</b> .....	<b>9</b>
Creación tabla usuarios en SQL.....	9
Restricciones de Dominio:.....	10
Creación tabla credencial en SQL.....	11
Restricciones de Dominio:.....	11
<b>Creacion de indices para mejoras de velocidad</b> .....	<b>12</b>
<b>Etap 2. Generación y carga de datos masivos con SQL puro</b> .....	<b>13</b>
Creacion del codigo de Carga:.....	13
Pruebas de Consistencia:.....	28
Conteos básicos.....	28
FK huérfanas.....	28
Cardinalidades del dominio.....	29
Validación de datos ingresados.....	30
<b>Etap 3. Consultas complejas y útiles a partir del CRUD inicial</b> .....	<b>32</b>
Consultas con JOIN.....	32
Consultas con y sin índice.....	34
2. Consulta con subconsulta.....	37
3. Consulta con group by + having (utiliza between).....	38
Medición comparativa con/sin índice en 3 consultas.....	40
Consulta con vista.....	41
<b>Etap 4 – Seguridad e Integridad:</b> .....	<b>42</b>
1. Script SQL con creación de usuario, permisos y vistas.....	42
CREACIÓN DEL USUARIO CON MÍNIMOS PRIVILEGIOS.....	42
DISEÑO DE 2 VISTAS QUE OCULTAN INFORMACIÓN SENSIBLE.....	43
ASIGNACIÓN DE PRIVILEGIOS MÍNIMOS.....	45
Demostración de los permisos del usuario:.....	46
2. Pruebas de integridad.....	49
3. Procedimiento SQL seguro (sin SQL dinámico).....	51
4. Prueba anti-inyección documentada.....	53
Intento de inyección malicioso:.....	54

<b>Etapa 5. Concurrencia y transacciones.....</b>	<b>56</b>
1. Simulación de Deadlocks.....	56
Como solucionar este deadlock:.....	59
2. Procedimiento almacenado en SQL.....	61
3 . READ COMMITTED y REPEATABLE READ.....	65
READ COMMITTED.....	65
REPEATABLE READ.....	66
Impacto de índices en entornos concurrentes.....	68
Creando indice para prueba:.....	70
Informe Etapa 5.....	72
<b>Anexo 1: Uso de la IA.....</b>	<b>73</b>
Etapa 1 – Modelado y Definición de Constraints:.....	73
Etapa 2. Generación de datos masivos con SQL puro.....	78
Etapa 3. Consultas complejas y útiles a partir del CRUD inicial.....	81
Etapa 4 – Seguridad e Integridad:.....	84
<b>Etapa 5. Concurrencia y transacciones.....</b>	<b>87</b>
Crear un procedimiento almacenado en SQL.....	88
<b>Fuentes de investigación:.....</b>	<b>91</b>

## Resumen ejecutivo

El proyecto desarrolla una base de datos centrada en usuarios y credenciales de acceso, garantizando integridad y seguridad.

Se modelan entidades, atributos y restricciones (PK, FK, UNIQUE, CHECK, dominios) y se generan datos masivos (10k–500k) con SQL puro.

Se implementan consultas avanzadas con JOIN, agregaciones, subconsultas y vistas para reportes.

La seguridad se refuerza con usuarios de privilegios mínimos, vistas que ocultan datos sensibles y consultas seguras en Java.

Finalmente, se simulan transacciones, bloqueos y niveles de aislamiento, aplicando estrategias de retry ante deadlocks.

## Reglas de negocio - Gestión de Usuarios y Credenciales

### Identificación única de usuarios

- Cada usuario debe tener un usuario único en el sistema.
- Cada usuario debe registrar un email único.

### Estado del usuario

- El campo activo solo puede tomar los valores: ACTIVO, INACTIVO o BLOQUEADO.
- Un usuario marcado como BLOQUEADO no puede iniciar sesión ni generar nuevas credenciales activas.

### Relación usuario–credencial

- Toda credencial debe estar asociada a un usuario existente (usuario\_id obligatorio).
- Un usuario puede tener una sola credencial activa a la vez.

### Estado de credenciales

- El campo estado de la credencial solo puede ser ACTIVO, INACTIVO

## Seguridad e integridad de datos

- La eliminación de un usuario implica la eliminación automática de sus credenciales asociadas (ON DELETE CASCADE).
- Las actualizaciones de id\_usuario se propagan a las credenciales (ON UPDATE CASCADE).

# Modelo Relacional y Constraints

## Tabla: usuario

- id\_usuario

**Tipo:** MEDIUMINT

**Restricciones:** PRIMARY KEY, AUTO\_INCREMENT

Para un máximo de 500 mil registros de prueba, el INT es demasiado grande y un desperdicio de espacio, el MEDIUM int nos permite ingresar estos 500 mil (con espacio de sobra) pero no es mucho más grande de lo necesario, como en el caso de INT

Se trata de la clave primaria de cada usuario, única e irrepetible. Al ser primary key nunca puede ser NULL y se utiliza un autoincrement para un manejo más simple de la base de datos y de los usuarios por parte de los administradores de dicha base de datos.

- Username VARCHAR(60)

**Restricciones:** NOT NULL

Creacion de nombre unicos a partir de la combinacion del nombre, apellido y un numero secuencial creado para cada usuario.

- **nombre**

**Tipo:** VARCHAR(100)

**Restricciones:** NOT NULL

Permite almacenar nombres completos incluso compuestos. Se obliga que sea NOT NULL ya que todos los usuarios deben tener su nombre.

- **apellido**

**Tipo:** VARCHAR(100)

**Restricciones:** NOT NULL

Misma lógica que en nombre, permite ingresar apellidos largos o compuestos.

- **email**

**Tipo:** VARCHAR(120)

**Restricciones:** NOT NULL, UNIQUE.

- **fecha\_registro**

**Tipo:** DATETIME

**Restricciones:** NOT NULL. No es posible ingresar una fecha futura.

**Restricciones:** NOT NULL, no se permite que no se ingrese este dato, debe estar siempre presente, ingresando el default como la fecha actual

- **Activo**

**Tipo:** BOOLEAN

**Restricciones:** NOT NULL con DEFAULT TRUE

Define el estado del usuario en el sistema. Se elige True por defecto

- **estado** VARCHAR(15)

**Restricciones:** NOT NULL DEFAULT 'ACTIVO',

Version textual simplificada del boolean activo, para una lectura mas simple por parte de los usuarios.

## **Tabla: credencial\_acceso**

- **id\_credencial**

**Tipo:** INT **Restricciones:** PRIMARY KEY, NOT NULL, AUTO\_INCREMENT.

Es el identificador único de cada credencial. El autoincremento simplifica la gestión de claves y asegura integridad de entidad.

- **usuario\_id**

**Tipo:** INT

**Restriccion:** NOT NULL, UNIQUE (garantiza la 1:1)

Clave foranea de la tabla, apunta a la tabla usuarios

- **estado**

**Tipo:** VARCHAR(10).

**Restricciones:** NOT NULL, CHECK

(estado IN ('ACTIVO','INACTIVO')).

Define el estado de la credencial. El CHECK limita los valores posibles, asegurando consistencia de dominio y evitando estados inválidos. (permite solamente ingresar 2 valores, Activo o Inactivo)

- **ultima\_sesion**

**Tipo:** TIMESTAMP.

**Restricciones:** NULL

Almacena la fecha y hora del último inicio de sesión. Se permite NULL para usuarios que nunca hayan ingresado. Útil para auditoría y control de actividad.

- **hash\_password** VARCHAR(255)

**Restricciones:** NOT NULL,

Almacena el resultado del algoritmo de hash aplicado a la contraseña del usuario (ej: bcrypt, Argon2, SHA-256 con sal). No se guarda la contraseña en texto plano.

- **salt** VARCHAR(64)

**Restricciones:** NULL

Se permite null para dar flexibilidad al sistema que almacena las credenciales, por ejemplo, si se utiliza Bcrypt, el mismo no necesita guardar el salt por separado. Ya se incluye en el hash que almacena.

- **ultimo\_cambio**

**Restricciones:** DATETIME



**Restriccion:** NULL

Almacena la fecha del último cambio que se realiza sobre la cuenta. Permite NULL para las cuentas que nunca se les realizó cambios.

- **requiere\_reset** BOOLEAN

**Restricciones:** NOT NULL DEFAULT FALSE

Un booleano que almacena si la cuenta requiere un reseteo de clave, se define not null, ya que almacena falso por defecto, porque una cuenta nueva no requiere el cambio de contraseña.

## Creación de Tablas

### Creación tabla usuarios en SQL

**ver 01\_esquema.sql**

```
CREATE TABLE usuario (  
  id_usuario INT NOT NULL AUTO_INCREMENT,  
  eliminado BOOLEAN NOT NULL DEFAULT FALSE,  
  username VARCHAR(60) NOT NULL,  
  nombre VARCHAR(100) NOT NULL,  
  apellido VARCHAR(100) NOT NULL,  
  email VARCHAR(120) NOT NULL,  
  fecha_registro DATETIME NOT NULL DEFAULT CURRENT_TIMESTAMP,  
  activo BOOLEAN NOT NULL DEFAULT TRUE,  
  estado VARCHAR(15) NOT NULL DEFAULT 'ACTIVO',  
  CONSTRAINT pk_usuario PRIMARY KEY (id_usuario),  
  CONSTRAINT uq_usuario_username UNIQUE (username),  
  CONSTRAINT uq_usuario_email UNIQUE (email),  
  CONSTRAINT ck_usuario_estado CHECK (estado IN ('ACTIVO','INACTIVO'))  
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_0900_ai_ci;
```

## **Restricciones de Dominio:**

- **Restricción de Primary Key**

```
CONSTRAINT pk_usuario PRIMARY KEY (id_usuario),
```

Declara id\_usuario como la clave primaria

- **Restricción uq\_usuario\_unico**

```
CONSTRAINT uq_usuario_unico UNIQUE (username);
```

Prohíbe que username se repita en otro campo de la base de datos, debe ser único

- **Restricción uq\_usuario\_email**

```
CONSTRAINT uq_usuario_email UNIQUE (email);
```

- **Restriccion check estado usuario**

```
CONSTRAINT ck_usuario_estado CHECK (estado IN ('ACTIVO', 'INACTIVO'))
```

No permite que el estado del usuario sea otro que: “activo” e inactivo”

## Creación tabla credencial en SQL

ver 01\_esquema.sql

```
CREATE TABLE credencial_acceso (  
  id_credencial INT NOT NULL AUTO_INCREMENT,  
  eliminado BOOLEAN NOT NULL DEFAULT FALSE,  
  usuario_id INT NOT NULL,  
  estado VARCHAR(15) NOT NULL,  
  ultima_sesion TIMESTAMP NULL,  
  hash_password VARCHAR(255) NOT NULL,  
  salt VARCHAR(64) NULL,  
  ultimo_cambio DATETIME NULL,  
  requiere_reset BOOLEAN NOT NULL DEFAULT FALSE,  
  CONSTRAINT pk_credencial PRIMARY KEY (id_credencial),  
  CONSTRAINT uq_credencial_usuario UNIQUE (usuario_id),  
  CONSTRAINT fk_credencial_usuario FOREIGN KEY (usuario_id)  
  REFERENCES usuario(id_usuario)  
  ON DELETE CASCADE ON UPDATE CASCADE,  
  CONSTRAINT ck_cred_estado CHECK (estado IN ('ACTIVO', 'INACTIVO'))  
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_0900_ai_ci;
```

### Restricciones de Dominio:

- **Restricción de Primary Key**

```
CONSTRAINT pk_credencial PRIMARY KEY (id_credencial),
```

Declara id\_credencial como clave primaria.

- **Restriccion credencial de usuario unica**

```
CONSTRAINT uq_credencial_usuario UNIQUE (usuario_id),
```

El usuario\_id debe ser unico e irrepitable.

- **Declaracion de clave foranea, on update y on delete**

```
CONSTRAINT fk_credencial_usuario FOREIGN KEY (usuario_id)
REFERENCES usuario(id_usuario)
ON DELETE CASCADE ON UPDATE CASCADE
```

Declara usuario\_id como clave foránea, que referencia id\_usuario de la tabla usuario. Realizando la conexión.

Al eliminarse un registro en usuario, automáticamente se eliminan todas las filas asociadas en credencial\_usuario.

Además, si el valor id\_usuario en la tabla usuario cambia, ese cambio se aplica automáticamente en la tabla credencial\_usuario.

- **Restriccion check estado usuario**

```
CONSTRAINT ck_cred_estado CHECK (estado IN ('ACTIVO', 'INACTIVO'))
```

## Creacion de indices para mejoras de velocidad

**Ver 04\_indices.sql**

```
CREATE INDEX ix_usuario_activo ON usuario(activo);
CREATE INDEX ix_usuario_estado ON usuario(estado);
CREATE INDEX ix_cred_estado ON credencial_acceso(estado);
```

Se crean indices para los estados de los usuarios y el estado de la credencial de los usuarios.

## Etaapa 2. Generación y carga de datos masivos con SQL puro

Ver 03\_carga\_masiva

Creacion del codigo de Carga:

```
1 • DROP DATABASE IF EXISTS gestion_usuarios_final;
2 • CREATE DATABASE IF NOT EXISTS gestion_usuarios_final
3     CHARACTER SET utf8mb4 COLLATE utf8mb4_0900_ai_ci;
4 • USE gestion_usuarios_final;
5
6 • SET @N := 100000; -- volumen de usuarios a generar
7
8 • DROP TABLE IF EXISTS credencial_acceso;
9 • DROP TABLE IF EXISTS usuario;
10
```

En este bloque de código SQL se define la estructura inicial del proyecto de base de datos `gestion_usuarios_final`. Primero se elimina la base de datos si ya existía (`DROP DATABASE IF EXISTS`) para evitar conflictos previos y luego se vuelve a crear (`CREATE DATABASE`) con el conjunto de caracteres `utf8mb4` y la colación `utf8mb4_0900_ai_ci`, que permiten almacenar textos con caracteres especiales, tildes y emojis. Posteriormente, la instrucción `USE gestion_usuarios_final;` selecciona esa base como la activa para las operaciones siguientes. La línea `SET @N := 100000;` crea una variable de usuario que indica la cantidad de registros (usuarios) que se generarán más adelante de forma masiva. Finalmente, se eliminan las tablas `credencial_acceso` y `usuario` si ya existen, dejando el entorno preparado para recrearlas desde cero y evitar errores de duplicación.

```

10
11 • ⊖ CREATE TABLE usuario (
12     id_usuario      INT NOT NULL AUTO_INCREMENT,
13     eliminado        BOOLEAN      NOT NULL DEFAULT FALSE,
14     username         VARCHAR(60)  NOT NULL,
15     nombre           VARCHAR(100) NOT NULL,
16     apellido         VARCHAR(100) NOT NULL,
17     email            VARCHAR(120) NOT NULL,
18     fecha_registro   DATETIME     NOT NULL DEFAULT CURRENT_TIMESTAMP,
19     activo           BOOLEAN      NOT NULL DEFAULT TRUE,
20     estado           VARCHAR(15)  NOT NULL DEFAULT 'ACTIVO', -- NUEVO
21     CONSTRAINT pk_usuario PRIMARY KEY (id_usuario),
22     CONSTRAINT uq_usuario_username UNIQUE (username),
23     CONSTRAINT uq_usuario_email  UNIQUE (email),
24     CONSTRAINT ck_usuario_estado CHECK (estado IN ('ACTIVO','INACTIVO'))
25 ) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_0900_ai_ci;
26

```

La instrucción CREATE TABLE usuario inicia la definición de la tabla, seguida por la lista de sus columnas y restricciones.

La columna id\_usuario INT NOT NULL AUTO\_INCREMENT define un identificador único para cada usuario, que se genera automáticamente en orden ascendente, funcionando como clave primaria.

La columna eliminado BOOLEAN NOT NULL DEFAULT FALSE indica si el registro fue eliminado lógicamente, permitiendo conservar los datos sin borrarlos físicamente.

Luego, username VARCHAR(60) NOT NULL, nombre VARCHAR(100) NOT NULL, apellido VARCHAR(100) NOT NULL y email VARCHAR(120) NOT NULL almacenan el nombre de usuario, nombre real, apellido y correo electrónico, respectivamente, todos obligatorios al no permitir valores nulos. La columna fecha\_registro DATETIME NOT NULL DEFAULT CURRENT\_TIMESTAMP guarda la fecha y hora en que se creó el registro, asignándola automáticamente al momento de inserción.

activo BOOLEAN NOT NULL DEFAULT TRUE indica si el usuario se encuentra activo en el sistema.

Por su parte, estado VARCHAR(15) NOT NULL DEFAULT 'ACTIVO' agrega una clasificación textual del estado del usuario (por ejemplo, 'ACTIVO' o 'INACTIVO').

A continuación, se definen las restricciones o reglas de integridad:

- CONSTRAINT pk\_usuario PRIMARY KEY (id\_usuario) establece que id\_usuario es la clave primaria, garantizando que cada usuario tenga un identificador único.
- CONSTRAINT uq\_usuario\_username UNIQUE (username) y CONSTRAINT uq\_usuario\_email UNIQUE (email) aseguran que no existan dos usuarios con el mismo nombre de usuario o correo electrónico.
- Finalmente, CONSTRAINT ck\_usuario\_estado CHECK (estado IN ('ACTIVO','INACTIVO')) restringe los valores posibles de la columna estado solo a esas dos opciones.

Por último, el fragmento ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4\_0900\_ai\_ci especifica que la tabla utilizará el motor de almacenamiento InnoDB (que permite claves foráneas y transacciones) y la codificación utf8mb4, que soporta todos los caracteres Unicode, incluidos tildes y emojis.

```

27 • CREATE TABLE credencial_acceso (
28     id_credencial INT NOT NULL AUTO_INCREMENT,
29     eliminado      BOOLEAN NOT NULL DEFAULT FALSE,
30     usuario_id     INT NOT NULL,
31     estado         VARCHAR(15) NOT NULL,
32     ultima_sesion  TIMESTAMP NULL,
33     hash_password  VARCHAR(255) NOT NULL,
34     salt           VARCHAR(64) NULL,
35     ultimo_cambio  DATETIME NULL,
36     requiere_reset BOOLEAN NOT NULL DEFAULT FALSE,
37     CONSTRAINT pk_credencial PRIMARY KEY (id_credencial),
38     CONSTRAINT uq_credencial_usuario UNIQUE (usuario_id),
39     CONSTRAINT fk_credencial_usuario FOREIGN KEY (usuario_id)
40         REFERENCES usuario(id_usuario)
41         ON DELETE CASCADE ON UPDATE CASCADE,
42     CONSTRAINT ck_cred_estado CHECK (estado IN ('ACTIVO','INACTIVO'))
43 ) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_0900_ai_ci;
44

```

En este bloque de código se crea la tabla credencial\_acceso, la cual complementa a la tabla usuario al guardar información relacionada con el acceso al sistema, como contraseñas cifradas, estados de cuenta y control de sesiones.

La instrucción CREATE TABLE credencial\_acceso inicia la creación de la tabla. La columna id\_credencial INT NOT NULL AUTO\_INCREMENT genera un identificador único y secuencial para cada credencial, que actúa como clave primaria.

eliminado BOOLEAN NOT NULL DEFAULT FALSE permite aplicar eliminaciones lógicas, sin borrar físicamente los registros.

usuario\_id INT NOT NULL funciona como clave foránea, relacionando cada credencial con el usuario correspondiente en la tabla usuario.

estado VARCHAR(15) NOT NULL almacena el estado de la credencial, pudiendo ser "ACTIVO" o "INACTIVO".

La columna ultima\_sesion TIMESTAMP NULL guarda la fecha y hora de la última vez que el usuario accedió al sistema.

hash\_password VARCHAR(255) NOT NULL contiene la contraseña cifrada mediante un algoritmo hash, mientras que salt VARCHAR(64) NULL guarda un valor aleatorio usado para reforzar la seguridad de la contraseña. ultimo\_cambio DATETIME NULL registra la última modificación de la clave, y requiere\_reset BOOLEAN NOT NULL DEFAULT FALSE indica si el usuario debe cambiar su contraseña al iniciar sesión.

En cuanto a las restricciones:

- CONSTRAINT pk\_credencial PRIMARY KEY (id\_credencial) define la clave primaria.
- CONSTRAINT uq\_credencial\_usuario UNIQUE (usuario\_id) asegura que cada usuario tenga solo una credencial asociada (relación 1:1).
- CONSTRAINT fk\_credencial\_usuario FOREIGN KEY (usuario\_id) REFERENCES usuario(id\_usuario) establece la relación con la tabla usuario, garantizando integridad referencial.
- Las cláusulas ON DELETE CASCADE ON UPDATE CASCADE indican que si se elimina o actualiza un usuario, su credencial se elimina o actualiza automáticamente, manteniendo la coherencia entre ambas tablas.
- CONSTRAINT ck\_cred\_estado CHECK (estado IN ('ACTIVO','INACTIVO')) valida que solo se admitan esos dos valores posibles en la columna estado.

Finalmente, la línea ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4\_0900\_ai\_ci especifica el motor de almacenamiento InnoDB, que permite claves foráneas y transacciones, y la codificación utf8mb4, ideal para textos con acentos y símbolos especiales



```
--
45 • CREATE INDEX ix_usuario_activo ON usuario(activo);
46 • CREATE INDEX ix_usuario_estado ON usuario(estado);
47 • CREATE INDEX ix_cred_estado ON credencial_acceso(estado);
48
```

En este bloque de código se crean índices que optimizan el rendimiento de las consultas sobre ciertas columnas de las tablas `usuario` y `credencial_acceso`.

La instrucción `CREATE INDEX ix_usuario_activo ON usuario(activo);` genera un índice sobre la columna `activo` de la tabla `usuario`. Esto permite que las búsquedas o filtros que determinen si un usuario está activo o no (por ejemplo, `WHERE activo = TRUE`) se ejecuten mucho más rápido, ya que el motor de la base de datos no necesita recorrer toda la tabla para encontrar los registros coincidentes.

Luego, `CREATE INDEX ix_usuario_estado ON usuario(estado);` crea otro índice sobre la columna `estado`, lo que mejora la eficiencia de las consultas que clasifican o filtran usuarios según su estado administrativo (por ejemplo, `ACTIVO` o `INACTIVO`).

Por último, `CREATE INDEX ix_cred_estado ON credencial_acceso(estado);` agrega un índice sobre la columna `estado` de la tabla `credencial_acceso`, acelerando las operaciones que consultan o agrupan credenciales por su estado (como `"ACTIVO"` o `"INACTIVO"`).

En conjunto, estos índices hacen que las consultas más comunes — especialmente las que implican filtros o condiciones en las columnas `activo` y `estado`— sean mucho más rápidas y eficientes, mejorando significativamente el rendimiento general de la base de datos.

```

50
51 DELIMITER $$
52
53 • -- 1) Si el usuario está INACTIVO, su credencial no puede quedar ACTIVA.
54 CREATE TRIGGER bi_credencial_no_activa_si_usuario_inactivo
55 BEFORE INSERT ON credencial_acceso
56 FOR EACH ROW
57 BEGIN
58     DECLARE v_estado_usuario VARCHAR(15);
59     SELECT estado INTO v_estado_usuario
60     FROM usuario
61     WHERE id_usuario = NEW.usuario_id;
62
63     IF v_estado_usuario = 'INACTIVO' AND NEW.estado = 'ACTIVO' THEN
64         -- Forzamos a INACTIVO para mantener coherencia
65         SET NEW.estado = 'INACTIVO';
66     END IF;
67 END$$
68
69 • CREATE TRIGGER bu_credencial_no_activa_si_usuario_inactivo
70 BEFORE UPDATE ON credencial_acceso
71 FOR EACH ROW
72 BEGIN
73     DECLARE v_estado_usuario VARCHAR(15);
74     SELECT estado INTO v_estado_usuario
75     FROM usuario
76     WHERE id_usuario = NEW.usuario_id;
77
78     IF v_estado_usuario = 'INACTIVO' AND NEW.estado = 'ACTIVO' THEN
79         SIGNAL SQLSTATE '45000'
80         SET MESSAGE_TEXT = 'No se puede activar la credencial de un usuario INACTIVO.';
81     END IF;
82 END$$

```

En este bloque se definen triggers para mantener la coherencia entre el estado del usuario y el de su credencial. Primero se cambia el delimitador con `DELIMITER $$` para poder escribir bloques `BEGIN ... END` que contienen varios ; internos. El trigger `CREATE TRIGGER`

`bi_credencial_no_activa_si_usuario_inactivo BEFORE INSERT ON credencial_acceso FOR EACH ROW` se ejecuta antes de insertar una credencial: dentro del bloque se declara `v_estado_usuario VARCHAR(15)`, se consulta el estado del usuario con `SELECT estado INTO v_estado_usuario FROM usuario WHERE id_usuario = NEW.usuario_id`; y, si el usuario está inactivo (`v_estado_usuario = 'INACTIVO'`) pero la nueva credencial viene activa (`NEW.estado = 'ACTIVO'`), se corrige forzándola a inactiva con `SET NEW.estado = 'INACTIVO'`; para mantener coherencia. El segundo trigger, `CREATE TRIGGER bu_credencial_no_activa_si_usuario_inactivo BEFORE UPDATE ON credencial_acceso`

FOR EACH ROW, se dispara antes de actualizar una credencial: repite la consulta del estado del usuario y, si intentan pasar la credencial a activa mientras el usuario sigue inactivo (`v_estado_usuario = 'INACTIVO' AND NEW.estado = 'ACTIVO'`), bloquea la operación lanzando un error con `SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT = 'No se puede activar la credencial de un usuario INACTIVO.'`. En ambos triggers, `NEW` referencia los valores que se están insertando/actualizando en la fila objetivo. Al finalizar cada definición se cierra el bloque con `END$$`.

```

68
69 • CREATE TRIGGER bu_credencial_no_activa_si_usuario_inactivo
70 BEFORE UPDATE ON credencial_acceso
71 FOR EACH ROW
72 BEGIN
73     DECLARE v_estado_usuario VARCHAR(15);
74     SELECT estado INTO v_estado_usuario
75     FROM usuario
76     WHERE id_usuario = NEW.usuario_id;
77
78     IF v_estado_usuario = 'INACTIVO' AND NEW.estado = 'ACTIVO' THEN
79         SIGNAL SQLSTATE '45000'
80         SET MESSAGE_TEXT = 'No se puede activar la credencial de un usuario INACTIVO.';
81     END IF;
82 END$$
83
84 -- 2) Si cambia el estado del usuario, sincronizamos la credencial.
85 • CREATE TRIGGER au_usuario_sync_estado_credencial
86 AFTER UPDATE ON usuario
87 FOR EACH ROW
88 BEGIN
89     IF NEW.estado <> OLD.estado THEN
90         UPDATE credencial_acceso
91         SET estado = CASE
92             WHEN NEW.estado = 'INACTIVO' THEN 'INACTIVO'
93             ELSE estado -- si el usuario pasa a ACTIVO, no forzamos; respetamos la cred actual
94         END
95         WHERE usuario_id = NEW.id_usuario;
96     END IF;
97 END$$
98
99 DELIMITER ;
100

```

En este fragmento se definen triggers adicionales que aseguran la coherencia entre los estados de la tabla `usuario` y la tabla `credencial_acceso`.

Primero se muestra el trigger `CREATE TRIGGER`

`bu_credencial_no_activa_si_usuario_inactivo` BEFORE UPDATE ON `credencial_acceso`, que se ejecuta antes de actualizar una credencial. Dentro del bloque `BEGIN ... END`, se declara la variable `v_estado_usuario` `VARCHAR(15)` y se obtiene el estado del usuario con la instrucción `SELECT estado INTO v_estado_usuario FROM usuario WHERE id_usuario = NEW.usuario_id`. Luego, si el usuario está inactivo (`v_estado_usuario = 'INACTIVO'`) y se intenta cambiar la credencial a activa (`NEW.estado = 'ACTIVO'`), se lanza un error mediante `SIGNAL SQLSTATE '45000'`, acompañado del mensaje 'No se puede activar la credencial de un usuario INACTIVO.', impidiendo así la operación.

A continuación, se define el trigger `CREATE TRIGGER`

au\_usuario\_sync\_estado\_credencial AFTER UPDATE ON usuario, que se ejecuta después de que se actualice un usuario. Su función es mantener sincronizado el estado de la credencial con el del usuario. Dentro del bloque, la condición IF NEW.estado <> OLD.estado THEN detecta si el estado del usuario cambió. Si es así, se actualiza la tabla credencial\_acceso con la instrucción UPDATE credencial\_acceso SET estado = CASE WHEN NEW.estado = 'INACTIVO' THEN 'INACTIVO' ELSE estado END WHERE usuario\_id = NEW.id\_usuario;. Esto significa que si el usuario pasa a INACTIVO, su credencial también se marca como INACTIVA, pero si el usuario vuelve a ACTIVO, no se modifica la credencial (se “respetar el estado actual”).

Finalmente, el bloque DELIMITER ; restaura el delimitador estándar, indicando el fin de la definición de los triggers. En conjunto, ambos mecanismos aseguran que los estados de usuario y credencial permanezcan siempre coherentes dentro del sistema.

```

101  /* =====
102  1) INSERCIÓN MASIVA USUARIOS
103  ===== */
104  INSERT INTO usuario (username, eliminado, nombre, apellido, email, fecha_registro, activo, estado)
105  SELECT
106      CONCAT(LOWER(nombre_sel.nombre), '.', LOWER(apellido_sel.apellido), '.', nums.n) AS username,
107      FALSE,
108      nombre_sel.nombre,
109      apellido_sel.apellido,
110      CONCAT(LOWER(nombre_sel.nombre), '.', LOWER(apellido_sel.apellido), '.', nums.n, '@example.com') AS email,
111      DATE_SUB(NOW(), INTERVAL (nums.n % 730) DAY),
112      CASE WHEN nums.n % 5 = 0 THEN FALSE ELSE TRUE END AS activo,
113      CASE WHEN nums.n % 7 = 0 THEN 'INACTIVO' ELSE 'ACTIVO' END AS estado -- patrón de ejemplo para poblar
114  FROM (
115      SELECT (d1.d*10000 + d2.d*1000 + d3.d*100 + d4.d*10 + d5.d) + 1 AS n
116      FROM
117          (SELECT 0 d UNION ALL SELECT 1 UNION ALL SELECT 2 UNION ALL SELECT 3 UNION ALL SELECT 4
118           UNION ALL SELECT 5 UNION ALL SELECT 6 UNION ALL SELECT 7 UNION ALL SELECT 8 UNION ALL SELECT 9) d1
119      CROSS JOIN
120          (SELECT 0 d UNION ALL SELECT 1 UNION ALL SELECT 2 UNION ALL SELECT 3 UNION ALL SELECT 4
121           UNION ALL SELECT 5 UNION ALL SELECT 6 UNION ALL SELECT 7 UNION ALL SELECT 8 UNION ALL SELECT 9) d2
122      CROSS JOIN
123          (SELECT 0 d UNION ALL SELECT 1 UNION ALL SELECT 2 UNION ALL SELECT 3 UNION ALL SELECT 4
124           UNION ALL SELECT 5 UNION ALL SELECT 6 UNION ALL SELECT 7 UNION ALL SELECT 8 UNION ALL SELECT 9) d3
125      CROSS JOIN
126          (SELECT 0 d UNION ALL SELECT 1 UNION ALL SELECT 2 UNION ALL SELECT 3 UNION ALL SELECT 4
127           UNION ALL SELECT 5 UNION ALL SELECT 6 UNION ALL SELECT 7 UNION ALL SELECT 8 UNION ALL SELECT 9) d4
128      CROSS JOIN
129          (SELECT 0 d UNION ALL SELECT 1 UNION ALL SELECT 2 UNION ALL SELECT 3 UNION ALL SELECT 4
130           UNION ALL SELECT 5 UNION ALL SELECT 6 UNION ALL SELECT 7 UNION ALL SELECT 8 UNION ALL SELECT 9) d5
131      WHERE (d1.d*10000 + d2.d*1000 + d3.d*100 + d4.d*10 + d5.d) + 1 <= @N
132  ) AS nums

```

En este bloque de código se realiza la inserción masiva de usuarios dentro de la tabla usuario, generando automáticamente miles de registros de ejemplo.

La instrucción INSERT INTO usuario (username, eliminado, nombre, apellido, email, fecha\_registro, activo, estado) especifica las columnas en las que se cargarán los datos. A continuación, el bloque SELECT genera dinámicamente los valores a insertar.

En `CONCAT(LOWER(nombre_sel.nombre), '.', LOWER(apellido_sel.apellido), '.', nums.n)` se construye el campo `username` uniendo el nombre, el apellido y un número identificador, todo en minúsculas y separados por puntos. La columna `eliminado` se fija en `FALSE`, ya que los registros se crean activos por defecto. Los campos `nombre` y `apellido` se toman de tablas auxiliares (`nombre_sel` y `apellido_sel`) que contienen combinaciones predefinidas. Para el correo electrónico, `CONCAT(LOWER(nombre_sel.nombre), '.', LOWER(apellido_sel.apellido), '.', nums.n, '@example.com')` genera direcciones únicas basadas en el mismo patrón.

La columna `fecha_registro` se asigna con `DATE_SUB(NOW(), INTERVAL (nums.n % 730) DAY)`, lo que crea fechas de registro aleatorias dentro de los últimos dos años.

Las columnas `activo` y `estado` se determinan mediante expresiones condicionales:

- `CASE WHEN nums.n % 5 = 0 THEN FALSE ELSE TRUE END AS activo` define que aproximadamente uno de cada cinco usuarios estará inactivo.
- `CASE WHEN nums.n % 7 = 0 THEN 'INACTIVO' ELSE 'ACTIVO' END AS estado` alterna el estado para simular distintos escenarios de uso.

El bloque `FROM (...) AS nums` genera una secuencia numérica del 1 hasta el valor definido en la variable `@N`, mediante una combinación de `CROSS JOIN` entre subconsultas con los dígitos del 0 al 9. Este método, conocido como “generador de números”, permite crear de forma rápida decenas o cientos de miles de filas sin usar una tabla preexistente.

En resumen, este script construye una gran cantidad de usuarios de manera automatizada, garantizando que cada uno tenga un `username`, email, y fecha de registro únicos, además de valores lógicos y variados en los campos `activo` y `estado`, facilitando pruebas de rendimiento y consistencia en la base de datos.

```

133 JOIN (
134     SELECT 1 id, 'Ana' nombre UNION ALL SELECT 2, 'Juan' UNION ALL SELECT 3, 'Sol' UNION ALL SELECT 4, 'Luis' UNION ALL
135     SELECT 5, 'Lucía' UNION ALL SELECT 6, 'Marcos' UNION ALL SELECT 7, 'Elena' UNION ALL SELECT 8, 'Diego' UNION ALL
136     SELECT 9, 'Carla' UNION ALL SELECT 10, 'Sofía' UNION ALL SELECT 11, 'Valentina' UNION ALL SELECT 12, 'Nicolás' UNION ALL
137     SELECT 13, 'Martina' UNION ALL SELECT 14, 'Agustín' UNION ALL SELECT 15, 'Camila' UNION ALL SELECT 16, 'Matías' UNION ALL
138     SELECT 17, 'Flores' UNION ALL SELECT 18, 'Tomás' UNION ALL SELECT 19, 'Paula' UNION ALL SELECT 20, 'Gabriel'
139 ) AS nombre_sel
140 ON nombre_sel.id = ((nums.n - 1) % 20) + 1
141 JOIN (
142     SELECT 1 id, 'Gómez' apellido UNION ALL SELECT 2, 'Pérez' UNION ALL SELECT 3, 'Martínez' UNION ALL SELECT 4, 'Ruiz' UNION ALL
143     SELECT 5, 'López' UNION ALL SELECT 6, 'Fernández' UNION ALL SELECT 7, 'Sánchez' UNION ALL SELECT 8, 'Vega' UNION ALL
144     SELECT 9, 'Navarro' UNION ALL SELECT 10, 'Castro' UNION ALL SELECT 11, 'Silva' UNION ALL SELECT 12, 'Torres' UNION ALL
145     SELECT 13, 'Romero' UNION ALL SELECT 14, 'Molina' UNION ALL SELECT 15, 'Ramos' UNION ALL SELECT 16, 'Herrera' UNION ALL
146     SELECT 17, 'Domínguez' UNION ALL SELECT 18, 'Gutiérrez' UNION ALL SELECT 19, 'Cabrera' UNION ALL SELECT 20, 'Acosta'
147 ) AS apellido_sel
148 ON apellido_sel.id = ((nums.n - 1) % 20) + 1
149 LEFT JOIN usuario u
150 ON u.username = CONCAT(LOWER(nombre_sel.nombre), '.', LOWER(apellido_sel.apellido), '.', nums.n)
151 WHERE u.id_usuario IS NULL;
152

```

En este tramo se arman “tablas en línea” de nombres y apellidos y se usan para generar los valores combinados sin depender de tablas físicas. Primero se define `JOIN ( ... ) AS nombre_sel`: es una lista de 20 nombres construida con `UNION ALL` y una clave `id` del 1 al

20; luego se la hace coincidir con cada número generado usando `ON nombre_sel.id = ((nums.n - 1) % 20) + 1`, que reparte cíclicamente los nombres. Lo mismo sucede con los apellidos en `JOIN ( ... ) AS apellido_sel` y su condición `ON apellido_sel.id = ((nums.n - 1) % 20) + 1`. Después, para evitar insertar duplicados si el script se corre más de una vez, se hace `LEFT JOIN usuario u` comparando el username calculado (`ON u.username = CONCAT(LOWER(nombre_sel.nombre), '.', LOWER(apellido_sel.apellido), '.', nums.n)`) y se filtra con `WHERE u.id_usuario IS NULL` para conservar solo las combinaciones que aún no existen. Así, cada fila generada obtiene un nombre y un apellido “rotando” uniformemente, y solo se insertan las nuevas.

```

-
3  /* =====
4  2) CREDENCIALES 1:1 (coherentes con usuario.estado)
5  ===== */
6  ● INSERT INTO credencial_acceso
7    (usuario_id, estado, ultima_sesion, eliminado, hash_password, salt, ultimo_cambio, requiere_reset)
8  SELECT
9    u.id_usuario,
10   -- Si el usuario está INACTIVO, la credencial queda INACTIVA; si no, patrón de estados
11   CASE
12     WHEN u.estado = 'INACTIVO' THEN 'INACTIVO'
13     WHEN u.id_usuario % 5 = 0 THEN 'INACTIVO'
14     ELSE 'ACTIVO'
15   END AS estado,
16   TIMESTAMP(
17     DATE_SUB(CURDATE(), INTERVAL (u.id_usuario % 365) DAY),
18     SEC_TO_TIME((u.id_usuario % 86400))
19   ),
20   FALSE,
21   UPPER(SHA2(CONCAT('pw:', u.id_usuario), 256)),
22   SUBSTRING(UPPER(SHA2(CONCAT('salt:', u.id_usuario), 256)), 1, 32),
23   DATE_SUB(NOW(), INTERVAL (u.id_usuario % 180) DAY),
24   (u.id_usuario % 20 = 0)
25 FROM usuario u
26 LEFT JOIN credencial_acceso c ON c.usuario_id = u.id_usuario
27 WHERE c.usuario_id IS NULL;
28

```

En este bloque se realiza la carga masiva de credenciales en la tabla `credencial_acceso`, estableciendo una relación 1:1 con cada usuario de la tabla `usuario`, y asegurando coherencia con el estado del mismo.

La instrucción `INSERT INTO credencial_acceso (usuario_id, estado, ultima_sesion, eliminado, hash_password, salt, ultimo_cambio, requiere_reset)` define las columnas donde se insertarán los datos. El bloque `SELECT` toma los valores directamente desde la tabla `usuario` (`FROM usuario u`), generando los campos calculados según reglas predefinidas.

En `CASE WHEN u.estado = 'INACTIVO' THEN 'INACTIVO' WHEN u.id_usuario % 5 = 0 THEN 'INACTIVO' ELSE 'ACTIVO' END AS estado`, se determina el estado de cada

credencial: si el usuario está inactivo, su credencial también lo estará; en caso contrario, se aplica un patrón que marca inactiva aproximadamente una de cada cinco credenciales.

El campo `ultima_sesion` se construye con `TIMESTAMP(DATE_SUB(CURDATE(), INTERVAL (u.id_usuario % 365) DAY), SEC_TO_TIME((u.id_usuario % 86400)))`, generando una fecha y hora de última sesión simuladas, distribuidas aleatoriamente dentro del último año.

La columna `eliminado` se fija en `FALSE`, indicando que las credenciales están activas por defecto.

Para el cifrado, `hash_password` se genera con `UPPER(SHA2(CONCAT('pw:', u.id_usuario), 256))`, aplicando la función hash SHA-256 a una cadena basada en el ID del usuario, mientras que `salt` se forma con

`SUBSTRING(UPPER(SHA2(CONCAT('salt:', u.id_usuario), 256)), 1, 32)`, creando un valor aleatorio de seguridad asociado a la contraseña.

La columna `ultimo_cambio` usa `DATE_SUB(NOW(), INTERVAL (u.id_usuario % 180) DAY)` para simular la fecha del último cambio de contraseña dentro de los últimos seis meses.

Finalmente, `(u.id_usuario % 20 = 0)` define que aproximadamente uno de cada veinte usuarios requiera restablecer su clave (`requiere_reset`).

El `LEFT JOIN credencial_acceso c ON c.usuario_id = u.id_usuario` garantiza que solo se creen credenciales nuevas para usuarios que aún no las tengan, gracias a la condición `WHERE c.usuario_id IS NULL`.

En conjunto, este bloque automatiza la generación de credenciales seguras, consistentes y sincronizadas con el estado de cada usuario, sin necesidad de ingresar los datos manualmente.

```

179  /* =====
180      3) PRUEBAS / VERIFICACIONES
181  ===== */
182
183  -- 3.1 Totales básicos
184  ● SELECT COUNT(*) AS total_usuarios FROM usuario;
185  ● SELECT COUNT(*) AS total_credenciales FROM credencial_acceso;
186
187  -- 3.2 Chequeo 1:1 (mismo conteo y sin huérfanas)
188  ● SELECT
189      (SELECT COUNT(*) FROM usuario)                AS usuarios,
190      (SELECT COUNT(*) FROM credencial_acceso)        AS credenciales,
191      (SELECT COUNT(*) FROM usuario u
192          LEFT JOIN credencial_acceso c ON c.usuario_id = u.id_usuario
193          WHERE c.usuario_id IS NULL)                AS usuarios_sin_cred,
194      (SELECT COUNT(*) FROM credencial_acceso c
195          LEFT JOIN usuario u ON u.id_usuario = c.usuario_id
196          WHERE u.id_usuario IS NULL)                AS cred_sin_usuario;
197
198  -- 3.3 PK*FK que coinciden (muestra)
199  ● SELECT u.id_usuario, u.username, u.estado AS estado_usuario,
200      c.id_credencial, c.usuario_id, c.estado AS estado_credencial
201  FROM usuario u
202  JOIN credencial_acceso c ON c.usuario_id = u.id_usuario
203  ORDER BY u.id_usuario
204  LIMIT 20;
205

```

En este bloque se encuentra la sección de pruebas y verificaciones, cuya función es comprobar que la carga de datos y las relaciones entre las tablas usuario y credencial\_acceso se realizaron de forma correcta y coherente.

En primer lugar, el apartado correspondiente a los totales básicos realiza un conteo general de registros en ambas tablas, contabilizando la cantidad total de usuarios y la cantidad total de credenciales almacenadas. Esto permite confirmar que el proceso de inserción masiva se completó satisfactoriamente y que ambas tablas contienen el volumen de datos esperado.

A continuación, en la verificación del modelo uno a uno (1:1), se ejecutan comprobaciones más detalladas. Se compara la cantidad total de registros de cada tabla y se buscan posibles inconsistencias, como usuarios que no tengan una credencial asociada o credenciales que no estén vinculadas a ningún usuario. De esta manera, se garantiza que cada usuario posea exactamente una credencial y que no existan registros “huérfanos” que rompan la integridad referencial entre las tablas.

Por último, la sección dedicada a las coincidencias entre claves primarias y foráneas muestra una muestra de registros vinculados. En ella se listan los primeros usuarios junto



con sus credenciales correspondientes, mostrando sus identificadores y estados. Esta revisión visual permite confirmar que las claves primarias de los usuarios coinciden correctamente con las claves foráneas de las credenciales y que los estados de ambas entidades se mantienen sincronizados.

En conjunto, este bloque de pruebas cumple la función de validar la correcta creación de los datos, la integridad de las relaciones y la coherencia lógica del modelo, asegurando que la base de datos funcione de manera consistente y sin errores estructurales.

```
-- 3.4 20 usuarios distintos (formato username)
SELECT id_usuario, username, nombre, apellido, email
FROM usuario
ORDER BY id_usuario
LIMIT 20;

-- 3.5 Validar cuantos username no cumplen el formato nombre.apellido.id
SELECT COUNT(*) AS fuera_de_formato
FROM usuario
WHERE username NOT REGEXP '^[a-z]+\.[a-z]+\.[0-9]+$';

-- 3.6 Distribuciones de estado/activo
SELECT estado, COUNT(*) AS cant FROM usuario GROUP BY estado;
SELECT activo, COUNT(*) AS cant FROM usuario GROUP BY activo;
SELECT estado, COUNT(*) AS cant FROM credencial_acceso GROUP BY estado;

-- 3.7 Pruebas negativas esperadas
-- a) UNIQUE en username/email
INSERT INTO usuario(username,nombre,apellido,email,fecha_registro,activo,estado)
VALUES ('ana.gomez.1','X','Y','ana.gomez.1@example.com',NOW(),1,'ACTIVO');

-- b) CHECK de estado inválido en usuario
INSERT INTO usuario(username,nombre,apellido,email,fecha_registro,activo,estado)
VALUES ('test.user.999999','Test','User','test.user.999999@example.com',NOW(),1,'SUSPENDIDO');
```

En este último bloque se incluyen las pruebas complementarias que permiten verificar aspectos de formato, distribución y validación de reglas dentro de la base de datos.

Primero, se muestra un conjunto de 20 usuarios distintos, donde se listan los campos principales —como el identificador, el nombre de usuario, el nombre real, el apellido y el correo electrónico—. Esta consulta sirve para obtener una vista rápida y representativa del contenido de la tabla, permitiendo comprobar visualmente que los datos se cargaron correctamente y que el formato de los nombres de usuario es coherente.

Luego, se realiza una validación de formato sobre el campo username. En este caso, se cuenta cuántos nombres de usuario no cumplen con la estructura establecida del tipo nombre.apellido.id. Para esto se emplea una expresión regular que verifica que todos los nombres estén en minúsculas, separados por puntos y finalizados con un número. Si el resultado del conteo es cero, significa que todos los usuarios cumplen con el formato esperado.

El siguiente apartado analiza las distribuciones de estado y actividad. Se generan agrupamientos que muestran cuántos usuarios están activos o inactivos, y cuántas credenciales se encuentran en cada estado. Esta información permite verificar que los datos insertados reflejan correctamente los patrones definidos durante la carga masiva y que los valores booleanos y de estado fueron asignados de forma proporcional.

Finalmente, se incluyen las pruebas negativas esperadas, cuyo propósito es comprobar el correcto funcionamiento de las restricciones de integridad. En la primera de ellas, se intenta insertar un registro duplicando el nombre de usuario y el correo electrónico, lo que debería generar un error al violar la restricción de unicidad. En la segunda, se prueba insertar un usuario con un estado no permitido, lo que activa el control de verificación (CHECK) definido en la tabla y provoca un error por valor inválido.

En conjunto, estas verificaciones finales no solo aseguran que los datos sigan las reglas de formato y distribución esperadas, sino que también confirman la robustez de las restricciones implementadas para proteger la integridad de la base de datos

```
31 -- c) CHECK de estado inválido en credencial
32 • INSERT INTO credencial_acceso(usuario_id,estado,hash_password)
33   VALUES (1,'SUSPENDIDO','X');
34
35 -- d) Trigger: no permitir credencial ACTIVA si usuario es INACTIVO
36 • UPDATE usuario SET estado='INACTIVO' WHERE id_usuario=2;
37 • UPDATE credencial_acceso SET estado='ACTIVO' WHERE usuario_id=2; -- debería lanzar error 45000
38
39 -- 3.8 Cascada ON DELETE: borrar usuario → borra su credencial
40 • SELECT COUNT(*) AS antes FROM credencial_acceso WHERE usuario_id=3;
41 • DELETE FROM usuario WHERE id_usuario=3;
42 • SELECT COUNT(*) AS despues FROM credencial_acceso WHERE usuario_id=3;
43
44 -- 3.9 Top 10 últimos logueos
45 • SELECT u.username, u.estado AS estado_usuario, c.estado AS estado_credencial, c.ultima_sesion
46   FROM credencial_acceso c
47   JOIN usuario u ON u.id_usuario = c.usuario_id
48   ORDER BY c.ultima_sesion DESC
49   LIMIT 10;
50
```

En este tramo final del bloque de pruebas se evalúan los mecanismos de control y las reglas de integridad más específicas de la base de datos.

Primero, se incluye una prueba de validación del estado en las credenciales, donde se intenta insertar un registro con un valor de estado no permitido. Esta acción está destinada a comprobar el correcto funcionamiento de la restricción CHECK definida en la tabla credencial\_acceso, que únicamente admite los valores “ACTIVO” o “INACTIVO”. Si la restricción opera correctamente, el sistema rechazará la inserción y devolverá un error, confirmando que la validación de dominio está activa.

A continuación, se verifica el comportamiento del trigger que impide inconsistencias entre el estado del usuario y el de su credencial. En este caso, se fuerza primero un cambio del

usuario a estado “INACTIVO” y luego se intenta activar su credencial. El trigger intercepta esta acción y lanza un error (de tipo 45000), asegurando que ningún usuario inactivo pueda tener una credencial activa. Este control automático garantiza la coherencia lógica entre ambas tablas sin necesidad de intervención manual.

Posteriormente, se comprueba la regla de eliminación en cascada (ON DELETE CASCADE). Para ello, se consulta la cantidad de credenciales existentes antes y después de eliminar un usuario determinado. Si la relación está correctamente configurada, la eliminación del usuario provoca automáticamente la eliminación de su credencial asociada. Esta prueba confirma que la clave foránea con comportamiento en cascada funciona correctamente, manteniendo la integridad referencial del modelo.

Por último, se realiza una consulta de los diez últimos inicios de sesión registrados. En esta verificación se listan los usuarios junto con sus estados y el momento de su última conexión, ordenados desde la más reciente hasta la más antigua. Esta prueba, además de servir como control de consistencia en los datos de auditoría, demuestra el uso práctico de la base de datos para obtener información operativa relevante.

En conjunto, este bloque final valida el correcto funcionamiento de las restricciones, los triggers y las dependencias en cascada, asegurando que la base de datos no solo mantenga la integridad estructural, sino también la coherencia lógica y funcional entre los distintos elementos del sistema.

```
250
251  -- 3.10 Sincronización automática ejemplo
252  •  UPDATE usuario SET estado='INACTIVO' WHERE id_usuario=4;
253  •  SELECT u.estado AS estado_usuario, c.estado AS estado_credencial
254  FROM usuario u JOIN credencial_acceso c ON c.usuario_id=u.id_usuario
255  WHERE u.id_usuario=4;
256
```

---

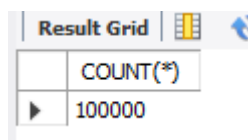
## Pruebas de Consistencia:

Ver 05\_consultas\_y\_explain.sql

### Conteos básicos

-- Total de usuarios

```
SELECT COUNT(*) FROM usuario;
```

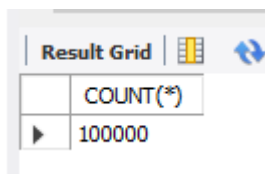


	COUNT(*)
▶	100000

Se verifica la creacion de 100 mil usuarios

-- Total de pedidos

```
SELECT COUNT(*) FROM credencial_acceso
```

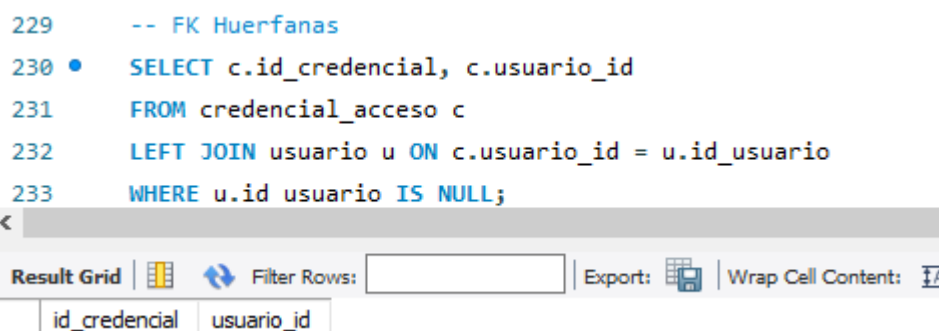


	COUNT(*)
▶	100000

Se verifica las 100 mil credenciales creadas

### FK huérfanas

```
SELECT c.id_credencial, c.usuario_id
FROM credencial_acceso c
LEFT JOIN usuario u ON c.usuario_id = u.id_usuario
WHERE u.id_usuario IS NULL;
```



```
229 -- FK Huerfanas
230 • SELECT c.id_credencial, c.usuario_id
231 FROM credencial_acceso c
232 LEFT JOIN usuario u ON c.usuario_id = u.id_usuario
233 WHERE u.id_usuario IS NULL;
```

	id_credencial	usuario_id
--	---------------	------------

Se puede ver que no existen FKs huérfanas

## Cardinalidades del dominio

```
-- Estados en usuario  
SELECT DISTINCT estado FROM usuario;
```

```
237      -- Estados en usuario  
238 •    SELECT DISTINCT estado FROM usuario;  
239
```

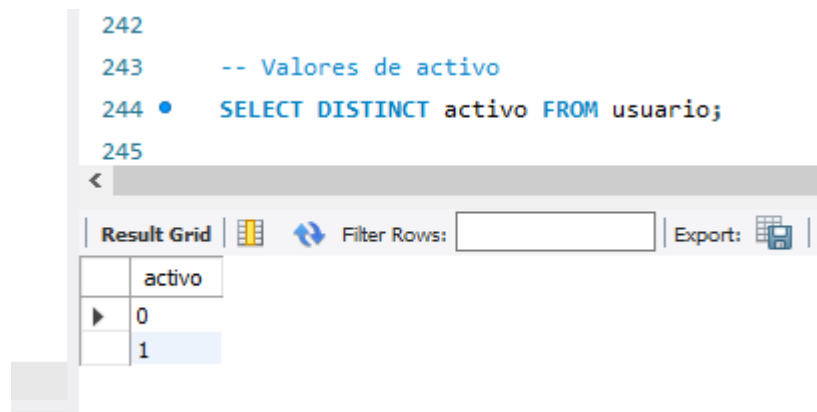
Result Grid		Filter Rows:	Export:	Wrap
estado				
▶	ACTIVO			
	INACTIVO			

```
-- Estados en credencial_acceso  
SELECT DISTINCT estado FROM credencial_acceso;
```

```
240      -- Estados en credencial_acceso  
241 •    SELECT DISTINCT estado FROM credencial_acceso;  
242
```

Result Grid		Filter Rows:	Export:	Wrap Cell C
estado				
▶	ACTIVO			
	inactivo			

```
-- Valores de activo
SELECT DISTINCT activo FROM usuario;
```



## Validación de datos ingresados

-- Se verifica que no se hayan creado usuarios sin nombre o apellido ingresado

```
SELECT COUNT(*) AS usuarios_sin_nombre_o_apellido
FROM usuario
WHERE (nombre IS NULL OR nombre = '')
      OR (apellido IS NULL OR apellido = '');
```

```

246 -- Se verifica que no se hayan creado usuarios sin nombre o apellido ingresado
247
248 • SELECT COUNT(*) AS usuarios_sin_nombre_o_apellido
249 FROM usuario
250 WHERE (nombre IS NULL OR nombre = '')
251        OR (apellido IS NULL OR apellido = '');
252

```

Result Grid		Filter Rows:	Export:	Wrap Cell Content:
	usuarios_sin_nombre_o_apellido			
▶	0			




## Etaapa 3. Consultas complejas y útiles a partir del CRUD inicial

Ver 05\_consultas\_y\_explain.sql

### Consultas con JOIN

1. Consulta en la que se pueden ver los últimos 10 usuarios que realizaron cambio de contraseña. Esta consulta podría usarse para saber si entre estos hay un determiando usuario.

```
5
6 • select u.nombre, u.apellido, c.ultimo_cambio from usuario u join
7   credencial_acceso c on u.id_usuario=c.usuario_id order by ultimo_cambio desc limit 10;
8
```

Result Grid			
Filter Rows: <input type="text"/>			
Export: 			
Wrap Cell Contents: 			
Fetch rows: 			
	nombre	apellido	ultimo_cambio
▶	Matías	Herrera	2025-10-19 20:37:45
	Florencia	Domínguez	2025-10-19 20:37:45
	Florencia	Domínguez	2025-10-19 20:37:45
	Florencia	Domínguez	2025-10-19 20:37:45
	Florencia	Domínguez	2025-10-19 20:37:45
	Florencia	Domínguez	2025-10-19 20:37:45
	Elena	Sánchez	2025-10-19 20:37:45
	Elena	Sánchez	2025-10-19 20:37:45
	Elena	Sánchez	2025-10-19 20:37:45
	Elena	Sánchez	2025-10-19 20:37:45



2. Consulta que selecciona nombre y apellido de usuarios que estuvieron activos en un rango de fechas. La intención de esta consulta es saber que usuarios ingresaron en el rango de fecha y horario particular. Podría usarse en caso de problemas de seguridad o un interés general.

```
1/  
18 -- 2  
19 • select u.nombre, u.apellido, c.ultima_sesion from usuario u join  
20 credencial_acceso c on u.id_usuario=c.usuario_id  
21 where c.ultima_sesion between "2025-05-15 04:24:00" and "2025-05-15 06:25:00";  
22
```

Result Grid			
Filter Rows:			
Export:   Wrap Cell Content:			
nombre	apellido	ultima_sesion	
Camila	Ramos	2025-05-15 04:24:12	
Camila	Ramos	2025-05-15 04:30:17	
Camila	Ramos	2025-05-15 04:36:22	
Lucía	López	2025-05-15 04:42:27	
Lucía	López	2025-05-15 04:48:32	
Lucía	López	2025-05-15 04:54:37	
Camila	Ramos	2025-05-15 05:00:42	
Camila	Ramos	2025-05-15 05:06:47	
Camila	Ramos	2025-05-15 05:12:52	
Lucía	López	2025-05-15 05:18:57	
Lucía	López	2025-05-15 05:25:02	
Matías	Herrera	2025-05-15 05:31:07	
Matías	Herrera	2025-05-15 05:37:12	
Matías	Herrera	2025-05-15 05:43:17	
Marcos	Ferná...	2025-05-15 05:49:22	
Marcos	Ferná...	2025-05-15 05:55:27	
Marcos	Ferná...	2025-05-15 06:01:32	
Matías	Herrera	2025-05-15 06:07:37	
Matías	Herrera	2025-05-15 06:13:42	
Marcos	Ferná...	2025-05-15 06:19:47	

# Consultas con y sin índice

1. Consulta “Usuarios que accedieron entre dos fechas”. Se utilizó una de las consultas join para registrar el tiempo de ejecución con y sin índice

Consulta sin índice: se ejecutó la consulta tres veces para registrar los tiempos

```
20 • select u.nombre, u.apellido, c.ultima_sesion from usuario u join
21 credencial_acceso c on u.id_usuario=c.usuario_id
22 where c.ultima_sesion between "2025-05-15 04:24:00" and "2025-05-15 06:25:00";
23 • show index from credencial_acceso;
24
25
```

nombre	apellido	ultima_sesion
Camilo	Ramos	2025-05-15 04:24:12
Camilo	Ramos	2025-05-15 04:30:17
Camilo	Ramos	2025-05-15 04:36:22
Lucia	Ulpez	2025-05-15 04:40:27
Lucia	Ulpez	2025-05-15 04:46:32
Lucia	Ulpez	2025-05-15 04:54:37
Camilo	Ramos	2025-05-15 05:00:42
Camilo	Ramos	2025-05-15 05:06:47
Camilo	Ramos	2025-05-15 05:12:52
Lucia	Ulpez	2025-05-15 05:18:57
Lucia	Ulpez	2025-05-15 05:25:02
María	Herrera	2025-05-15 05:31:07
María	Herrera	2025-05-15 05:37:12
María	Herrera	2025-05-15 05:43:17
Marcos	Fernández	2025-05-15 05:49:22
Marcos	Fernández	2025-05-15 05:55:27
Marcos	Fernández	2025-05-15 06:01:32
María	Herrera	2025-05-15 06:07:37
María	Herrera	2025-05-15 06:13:42
Marcos	Fernández	2025-05-15 06:19:47

#	Time	Action	Message	Duration / Fetch
84	20:59:39	show index from credencial_acceso	3 row(s) returned	0.000 sec / 0.000 sec
85	21:00:11	select u.nombre, u.apellido, c.ultima_sesion from usuario u join credencial_acceso c on u.id_usuario=c.usuario_id where c.ultima_sesion between "2025-05-15 04:24:00" and "2025-05-15 06:25:00";	20 row(s) returned	0.062 sec / 0.000 sec
86	21:00:12	select u.nombre, u.apellido, c.ultima_sesion from usuario u join credencial_acceso c on u.id_usuario=c.usuario_id where c.ultima_sesion between "2025-05-15 04:24:00" and "2025-05-15 06:25:00";	20 row(s) returned	0.047 sec / 0.000 sec
87	21:00:14	select u.nombre, u.apellido, c.ultima_sesion from usuario u join credencial_acceso c on u.id_usuario=c.usuario_id where c.ultima_sesion between "2025-05-15 04:24:00" and "2025-05-15 06:25:00";	20 row(s) returned	0.062 sec / 0.000 sec

Tiempos sin indice:

77	08:58:44	select u.nombre, u.apellido, c.ultima_sesion from usuario u join credencial_acceso c on u.id_usuario=c.usuario_id where c.ultima_sesion between "2025-05-15 04:24:00" and "2025-05-15 06:25:00";	0.062 sec / 0.000 sec	20 row(s) returned
----	----------	--	-----------------------	--------------------

Explain de la consulta donde no consta el uso de índices específicos

```
25
26 • EXPLAIN select u.nombre, u.apellido, c.ultima_sesion from usuario u join
27 credencial_acceso c on u.id_usuario=c.usuario_id
28 where c.ultima_sesion between "2025-05-15 04:24:00" and "2025-05-15 06:25:00";
29
```

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	c	NULL	ALL	uq_credencial_usuario	NULL	NULL	NULL	99112	11.11	Using where
1	SIMPLE	u	NULL	eq_ref	PRIMARY	PRIMARY	4	gestion_usuarios.c.usuario_id	1	100.00	NULL

**EXPLAIN ANALYZE** de la consulta sin índice:

```
EXPLAIN ANALYZE select u.nombre, u.apellido, c.ultima_sesion from usuario u join
credencial_acceso c on u.id_usuario=c.usuario_id
where c.ultima_sesion between "2025-05-15 04:24:00" and "2025-05-15 06:25:00";
```

```
'-> Nested loop inner join (cost=14022 rows=11018) (actual
time=8.41..59.9 rows=20 loops=1)\n      -> Filter: (c.ultima_sesion
between \'2025-05-15 04:24:00\' and \'2025-05-15 06:25:00\')
(cost=10166 rows=11018) (actual time=8.4..59.9 rows=20 loops=1)\n
-> Table scan on c (cost=10166 rows=99176) (actual time=0.277..18.1
rows=100000 loops=1)\n      -> Single-row index lookup on u using
PRIMARY (id_usuario=c.usuario_id) (cost=0.25 rows=1) (actual
time=0.00232..0.00236 rows=1 loops=20)\n'
```

Se puede ver la falta de índice y el tiempo que le lleva la lectura de la tabla.

Creación y muestra de índice en la columna ultima\_sesion de la tabla credencial\_acceso

```
46 • create index idx_ultima_sesion on credencial_acceso(ultima_sesion);
47 • show index from credencial_acceso;
```

48

Table	Non_unique	Key_name	Seq_in_index	Column_name	Collation	Cardinality	Sub_part	Packed	Null	Index_type	Comment	Index_comment	Visible	Expression
crede...	0	PRIMARY	1	id_credencial	A	99112	HULL	HULL		BTREE			YES	HULL
crede...	0	uq_credencial_usuario	1	usuario_id	A	97109	HULL	HULL		BTREE			YES	HULL
crede...	1	ix_cred_estado	1	estado	A	1	HULL	HULL		BTREE			YES	HULL
crede...	1	idx_ultima_sesion	1	ultima_sesion	A	99112	HULL	HULL	YES	BTREE			YES	HULL

Consulta con índice: tiempo de ejecución 0.00 segundos en las 3 repeticiones

```

20 • select u.nombre, u.apellido, c.ultima_sesion from usuario u join
21 credencial_acceso c on u.id_usuario=c.usuario_id
22 where c.ultima_sesion between "2025-05-15 04:24:00" and "2025-05-15 06:25:00";
23 • show index from credencial_acceso;
24
25
26

```

The screenshot shows a database query execution interface. The top part displays a SQL query with three lines: a JOIN query, a WHERE clause with a time range, and a SHOW INDEX command. Below the query, a 'Result Grid' shows 20 rows of data with columns 'nombre', 'apellido', and 'ultima\_sesion'. The bottom part shows an 'Output' window with an 'Action Output' tab, displaying a log of the query execution steps and their durations.

#	Time	Action	Message	Duration / Fetch
89	21:05:41	show index from credencial_acceso	4 row(s) returned	0.016 sec / 0.000 sec
90	21:06:00	select u.nombre, u.apellido, c.ultima_sesion from usuario u join credencial_acceso c on u.id_usuario=c.usuario_id where c.ultima...	20 row(s) returned	0.000 sec / 0.000 sec
91	21:06:02	select u.nombre, u.apellido, c.ultima_sesion from usuario u join credencial_acceso c on u.id_usuario=c.usuario_id where c.ultima...	20 row(s) returned	0.000 sec / 0.000 sec
92	21:06:04	select u.nombre, u.apellido, c.ultima_sesion from usuario u join credencial_acceso c on u.id_usuario=c.usuario_id where c.ultima...	20 row(s) returned	0.000 sec / 0.000 sec

Icon	Time	Query	Duration	Rows
✓	81 09:01:44	select u.nombre, u.apellido, c.ultima_sesion from usuario u join credencial_acceso c...	0.000 sec / 0.000 sec	20 row(s) returned

Se puede ver que el tiempo de ejecución de una consulta con índice es menor que si la tabla no tuviera índice. Por lo tanto se cumple el objetivo de aumentar el rendimiento de la base de datos. Además el explain de la consulta lo demuestra: el campo key utiliza el índice creado.

```

25
26 • EXPLAIN select u.nombre, u.apellido, c.ultima_sesion from usuario u join
27 credencial_acceso c on u.id_usuario=c.usuario_id
28 where c.ultima_sesion between "2025-05-15 04:24:00" and "2025-05-15 06:25:00";
29

```

The screenshot shows a database query execution interface. The top part displays a SQL query with an EXPLAIN statement. Below the query, a 'Result Grid' shows the execution plan for the query.

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	c		range	uq_credencial_usuario_idx_ultima_sesion	idx_ultima_sesion	5		20	100.00	Using index condition
1	SIMPLE	u		eq_ref	PRIMARY		4	gestion_usuarios.c.usuario_id	1	100.00	

## EXPLAIN ANALYZE de la consulta con indice:

```
88 09:05:59 EXPLAIN ANALYZE select u.nombre, u.apellido, c.ultima_sesion from usuario u join credencial_acceso c on u.id_usuario=c.usuario_id where c.... 0.000 sec / 0.000 sec 1 row(s) returned
```

```
0.000 sec / 0.000 sec 1 row(s) returned
```

'-> Nested loop inner join (cost=16.3 rows=20) (actual time=0.195..0.223 rows=20 loops=1)\n -> Index range scan on c using idx\_ultima\_sesion over ('2025-05-15 04:24:00' <= ultima\_sesion <= '2025-05-15 06:25:00'), with index condition: (c.ultima\_sesion between '2025-05-15 04:24:00' and '2025-05-15 06:25:00') (cost=9.26 rows=20) (actual time=0.188..0.191 rows=20 loops=1)\n -> Single-row index lookup on u using PRIMARY (id\_usuario=c.usuario\_id) (cost=0.255 rows=1) (actual time=0.0014..0.00141 rows=1 loops=20)\n'

Ahora la consulta utiliza el indice

## 2. Consulta con subconsulta.

### Asistencia de IA ChatGPT

Consulta: *“Usuarios que requieren reset de contraseña”*. Esta consulta tiene una subconsulta en la cláusula *WHERE*. Este tipo de consulta puede usarse como una ayuda para recordar cuáles usuarios deben cambiar la contraseña. La interacción con la IA se puede ver en el anexo.

```

77 • SELECT
78   u.username,
79   u.email,
80   c.requiere_reset,
81   c.ultimo_cambio
82 FROM usuario u
83 JOIN credencial_acceso c ON u.id_usuario = c.usuario_id
84 WHERE id_usuario in (select id_usuario from credencial_acceso where requiere_reset=1);
85

```

Result Grid

username	email	requiere_reset	ultimo_cambio
carla.ramos.96035	carla.ramos.96035@example.com	0	2025-10-14 20:37:45
carla.ramos.5035	carla.ramos.5035@example.com	0	2025-10-12 20:37:45
carla.ramos.13035	carla.ramos.13035@example.com	0	2025-10-12 20:37:45
carla.ramos.23035	carla.ramos.23035@example.com	0	2025-10-11 20:37:45
carla.ramos.33035	carla.ramos.33035@example.com	0	2025-10-10 20:37:45
carla.ramos.43035	carla.ramos.43035@example.com	0	2025-10-09 20:37:45
carla.ramos.53035	carla.ramos.53035@example.com	0	2025-10-08 20:37:45
carla.ramos.63035	carla.ramos.63035@example.com	0	2025-10-07 20:37:45
carla.ramos.73035	carla.ramos.73035@example.com	0	2025-10-06 20:37:45
carla.ramos.83035	carla.ramos.83035@example.com	0	2025-10-05 20:37:45
carla.ramos.93035	carla.ramos.93035@example.com	0	2025-10-04 20:37:45
carla.ramos.4035	carla.ramos.4035@example.com	0	2025-10-03 20:37:45
carla.ramos.14035	carla.ramos.14035@example.com	0	2025-10-02 20:37:45
carla.ramos.24035	carla.ramos.24035@example.com	0	2025-10-01 20:37:45
carla.ramos.34035	carla.ramos.34035@example.com	0	2025-09-30 20:37:45
carla.ramos.44035	carla.ramos.44035@example.com	1	2025-09-29 20:37:45
carla.ramos.54035	carla.ramos.54035@example.com	0	2025-09-28 20:37:45
carla.ramos.64035	carla.ramos.64035@example.com	0	2025-09-27 20:37:45
carla.ramos.74035	carla.ramos.74035@example.com	0	2025-09-26 20:37:45
carla.ramos.84035	carla.ramos.84035@example.com	0	2025-09-25 20:37:45

Output

Step	Time	Message	Duration / Fetch
94	21.1237	SELECT u.username, u.email, c.requiere_reset, c.ultimo_cambio FROM usuario u JOIN credencial_acceso c ON u.id_usuario = c.usuario_id	1000 rows returned
95	21.1238	SELECT u.username, u.email, c.requiere_reset, c.ultimo_cambio FROM usuario u JOIN credencial_acceso c ON u.id_usuario = c.usuario_id	0.000 sec / 0.000 sec
96	21.1239	SELECT u.username, u.email, c.requiere_reset, c.ultimo_cambio FROM usuario u JOIN credencial_acceso c ON u.id_usuario = c.usuario_id	0.015 sec / 0.000 sec
97	21.1240	SELECT u.username, u.email, c.requiere_reset, c.ultimo_cambio FROM usuario u JOIN credencial_acceso c ON u.id_usuario = c.usuario_id	0.000 sec / 0.000 sec

Esta consulta, a pesar de no tener índice da un valor muy bajo de tiempo de respuesta.

Explain de la consulta

```

73 -- Usuarios que requieren reset de contraseña:
74
75 • Explain SELECT
76   u.username,
77   u.email,
78   c.requiere_reset,
79   c.ultimo_cambio
80 FROM usuario u
81 JOIN credencial_acceso c ON u.id_usuario = c.usuario_id
82 WHERE id_usuario in (select id_usuario from credencial_acceso where requiere_reset=1);
83

```

Result Grid

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	c		ALL	uq_credencial_usuario				99112	100.00	
1	SIMPLE	u		eq_ref		PRIMARY	4	gestion_usuarios.c.usuario_id	1	100.00	
1	SIMPLE	credencial_acceso		ALL					99112	10.00	Using where; FirstMatch(u); Using join buffer (hash join)

### 3. Consulta con group by + having (utiliza between)

Consulta: “Usuarios registrados en 1 dia”. La consulta sirve para obtener información de cuantas credenciales (usuarios) dentro de una fecha específica hubo activas e inactivas, en este caso el 20 de mayo de 2025. Que a su vez se hayan activado al menos una vez.

```

116
117 -- =====
118 -- consulta con group by + having
119 -- ¿Qué usuarios registrados en los últimos 30 días ingresaron mas de dos veces?
120
121
122 • select count(*), estado from credencial_acceso where ultima_sesion between '2025-05-20 00:00:00' and '2025-05-20 23:59:59'
123   GROUP BY estado having count(estado) > 273 ;
124
125

```

Explain de la consulta

```
123
124 * explain select count(*), estado from credencial_acceso where ultima_sesion between '2025-05-20 00:00:00' and '2025-05-20 23:59:59'
125 GROUP BY estado having count(estado) > 273 ;
126
127
```

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	credencial_acceso		index	ix_cred_estado	ix_cred_estado	62		99112	11.11	Using where

Se crea un índice en la columna ultima\_sesion de la tabla credencial\_acceso:

```
128 * create index idx_ultima_sesion on credencial_acceso(ultima_sesion);
129 * show index from credencial_acceso;
130
```

Table	Non_unique	Key_name	Seq_in_index	Column_name	Collation	Cardinality	Sub_part	Packed	Null	Index_type	Comment	Index_comment	Visible	Expression
crede...	0	PRIMARY	1	id_credencial	A	99112				BTREE			YES	
crede...	0	uq_credencial_usuario	1	usuario_id	A	97109				BTREE			YES	
crede...	1	ix_cred_estado	1	estado	A	1				BTREE			YES	
crede...	1	idx_ultima_sesion	1	ultima_sesion	A	99112			YES	BTREE			YES	

Al ejecutar la consulta se observan valores repetidos por debajo de la milésima de segundo.

```
123
124 * select count(*), estado from credencial_acceso where ultima_sesion between '2025-05-20 00:00:00' and '2025-05-20 23:59:59'
125 GROUP BY estado having count(estado) > 273 ;
126
```

#	Time	Action	Message	Duration / Fetch
10	16.01.16	show index from credencial_acceso	4 row(s) returned	0.000 sec / 0.000 sec
11	16.02.17	select count(*), estado from credencial_acceso where ultima_sesion between '2025-05-20 00:00:00' and '2025-05-20 23:59:59' GROUP BY estado having ...	1 row(s) returned	0.000 sec / 0.000 sec
12	16.02.21	select count(*), estado from credencial_acceso where ultima_sesion between '2025-05-20 00:00:00' and '2025-05-20 23:59:59' GROUP BY estado having ...	1 row(s) returned	0.000 sec / 0.000 sec
13	16.02.22	select count(*), estado from credencial_acceso where ultima_sesion between '2025-05-20 00:00:00' and '2025-05-20 23:59:59' GROUP BY estado having ...	1 row(s) returned	0.000 sec / 0.000 sec

## Medición comparativa con/sin índice en 3 consultas

Sobre las mediciones comparativas en las tablas mostradas se puede decir:

### 1) El uso de índice mejoró el rendimiento de las consultas.

- La consulta *“Usuarios que accedieron entre dos fechas”*; la mediana pasó de ser 0,062 segundos sin índice a ser de 0,0 segundos con índice.
- En la consulta *“Usuarios registrados en los últimos 30 días”* la mediana fue de 0,125 segundos sin índice y los valores con índice fueron de 0,0 segundos.

### 2) Se obtuvo un valor de 0,0 segundos en la tabla *“Usuarios que requieren reset de contraseña”* sin índice.

Tabla de tiempos de consultas con y sin índice

Nombre de la consulta	Tiempo sin índice (segundos)	Tiempo con índice (segundos)
<i>“Usuarios que accedieron entre dos fechas”</i>	Mediana = 0,062	0,0
<i>“Usuarios registrados en los últimos 30 días”</i>	Mediana = 0,125	0,0
<i>“Usuarios que requieren reset de contraseña”</i>	0,0	0,0



## Consulta con vista

Consulta “Usuario requiere reset de contraseña”. Podría ser útil usar una vista de esta tabla en contextos donde muchos usuarios la necesiten acceder a la misma información, como en un call center de una empresa que da servicios de internet o plataformas de tv.

Se puede chequear si un usuario (id\_usuario =2 en la imagen) requiere reset de contraseña de manera rápida.

```
66 • create view vista_usuarios_requiere_reset as
67 select u.id_usuario, u.nombre, u.apellido, u.email, c.requiere_reset
68 from usuario u join credencial_acceso c on u.id_usuario=c.usuario_id ;
69
70 • select id_usuario, requiere_reset from vista_usuarios_requiere_reset where id_usuario = 2;
71
72
```

Result Grid		Filter Rows:	Export:	Wrap Cell Content:
id_usuario	requiere_reset			
2	0			

## Etapa 4 – Seguridad e Integridad:

Ver 06\_seguridad\_vistas\_seguras.sql

### 1. Script SQL con creación de usuario, permisos y vistas.

```
USE gestion_usuarios_final;
```

#### CREACIÓN DEL USUARIO CON MÍNIMOS PRIVILEGIOS

-- Se crea un usuario de aplicación que solo puede conectarse desde la máquina local.

-- 1. CREACIÓN DEL USUARIO CON MÍNIMOS PRIVILEGIOS

-- Se crea un usuario de aplicación que solo puede conectarse desde la máquina local.

```
CREATE USER 'usuario_app'@'localhost' IDENTIFIED BY 'password_segura';
```

```
GRANT SELECT ON gestion_usuarios_final.* TO 'usuario_app'@'localhost';
```

```
GRANT SELECT ON gestion_usuarios_final.v_credenciales_no_sensible TO 'usuario_app'@'localhost';
```

```
FLUSH PRIVILEGES;
```

```
-- 1. CREACIÓN DEL USUARIO CON MÍNIMOS PRIVILEGIOS
-- Se crea un usuario de aplicación que solo puede conectarse desde la máquina local.
CREATE USER 'usuario_app'@'localhost' IDENTIFIED BY 'password_segura';
GRANT SELECT ON gestion_usuarios_final.* TO 'usuario_app'@'localhost';
GRANT SELECT ON gestion_usuarios_final.v_credenciales_no_sensible TO 'usuario_app'@'localhost';
FLUSH PRIVILEGES;
```

## DISEÑO DE 2 VISTAS QUE OCULTAN INFORMACIÓN SENSIBLE

```
-- VISTA 1: v_usuarios_publico

-- Oculta: 'eliminado', 'fecha_registro' (información de auditoría/administración).

CREATE OR REPLACE VIEW v_usuarios_publico AS

SELECT

    id_usuario,

    username,

    nombre,

    apellido,

    email,

    activo

FROM

    usuario;
```

```
1  -- VISTA 1: v_usuarios_publico
2  -- Oculta: 'eliminado', 'fecha_registro' (información de auditoría/administración).
3  • CREATE OR REPLACE VIEW v_usuarios_publico AS
4  SELECT
5      id_usuario,
6      username,
7      nombre,
8      apellido,
9      email,
0      activo
1  FROM
2      usuario;
3
4  • select * from v_usuarios_publico;
5
```

**Se puede ver que oculta la información sensible**

**Vista 2:**

```
-- VISTA 2: v_credenciales_no_sensible

-- Oculta: 'hash_password' y 'salt' (información criptográfica
sensible para autenticación).

CREATE OR REPLACE VIEW v_credenciales_no_sensible AS

SELECT

    usuario_id,

    estado,

    ultima_sesion,

    ultimo_cambio,

    requiere_reset

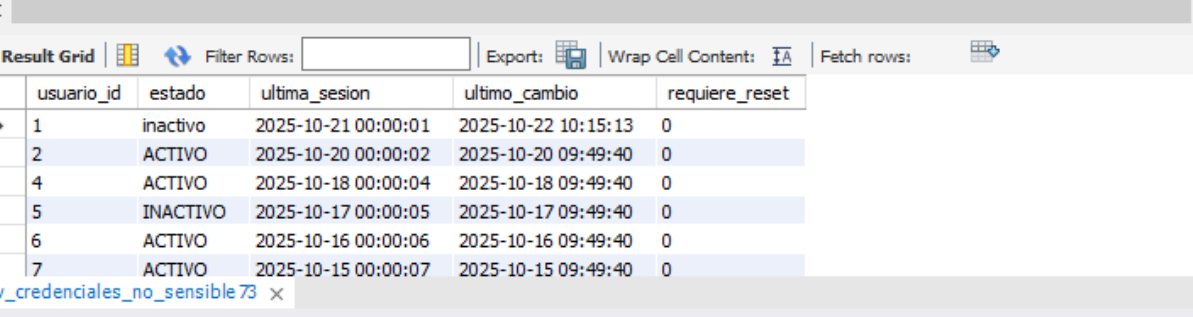
FROM

    credencial_acceso;
```

```

286 -- VISTA 2: v_credenciales_no_sensible
287 -- Oculta: 'hash_password' y 'salt' (información criptográfica sensible para autenticación).
288 • CREATE OR REPLACE VIEW v_credenciales_no_sensible AS
289 SELECT
290     usuario_id,
291     estado,
292     ultima_sesion,
293     ultimo_cambio,
294     requiere_reset
295 FROM
296     credencial_acceso;
297
298 • select * from v_credenciales_no_sensible;

```



	usuario_id	estado	ultima_sesion	ultimo_cambio	requiere_reset
1	1	inactivo	2025-10-21 00:00:01	2025-10-22 10:15:13	0
2	2	ACTIVO	2025-10-20 00:00:02	2025-10-20 09:49:40	0
4	4	ACTIVO	2025-10-18 00:00:04	2025-10-18 09:49:40	0
5	5	INACTIVO	2025-10-17 00:00:05	2025-10-17 09:49:40	0
6	6	ACTIVO	2025-10-16 00:00:06	2025-10-16 09:49:40	0
7	7	ACTIVO	2025-10-15 00:00:07	2025-10-15 09:49:40	0

Se puede observar que la vista oculta la información sensible acerca de las credenciales

## ASIGNACIÓN DE PRIVILEGIOS MÍNIMOS

El usuario con permisos limitados solo puede ver información

-- El usuario solo tiene permisos de SELECT sobre las vistas.

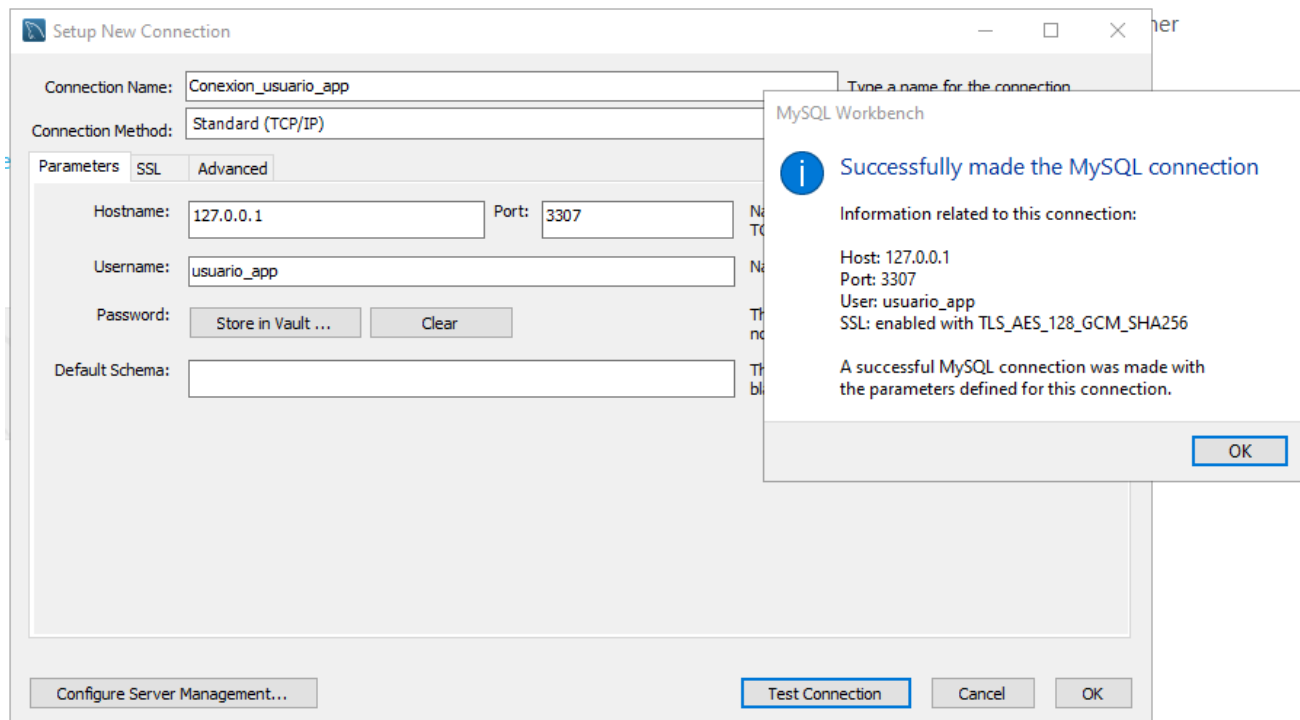
```
GRANT SELECT ON gestion_usuarios.v_usuarios_publico TO 'usuario_app'@'localhost';
```

```
GRANT SELECT ON gestion_usuarios.v_credenciales_no_sensible TO 'usuario_app'@'localhost';
```

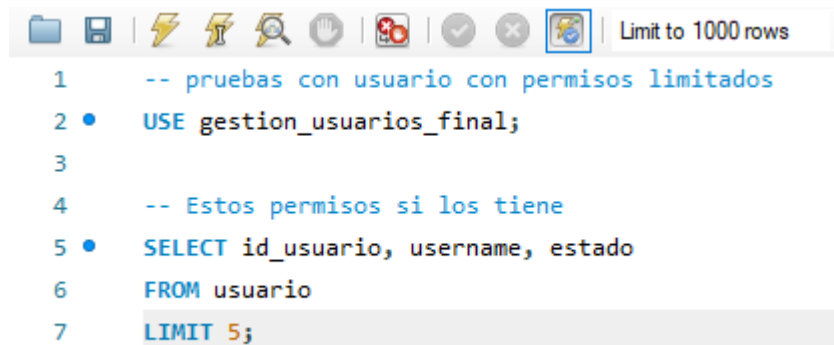
```
FLUSH PRIVILEGES;
```

## Demostración de los permisos del usuario:

Se crea nueva conexión con el usuario con permisos limitados



Se prueba que puede realizar un SELECT y ver informacion



Result Grid			
Filter Rows: <input type="text"/>			
Edit:			
Export/Import:			
	id_usuario	username	estado
▶	1	valentina.silva.90031	ACTIVO
	2	valentina.silva.9131	ACTIVO
	3	valentina.silva.19131	INACTIVO
	4	valentina.silva.29131	ACTIVO
	5	valentina.silva.39131	ACTIVO
*	NULL	NULL	NULL

usuario 1 x			
Output			
Action Output			
#	Time	Action	Message
✓ 1	13:25:27	USE gestion_usuarios_final	0 row(s) affected
✓ 2	13:25:27	SELECT id_usuario, username, estado FROM usuario LIMIT 5	5 row(s) returned

## Se intenta insertar datos con el usuario

```
-- no es posible insertar con este usuario ningun dato
INSERT INTO usuario (username, nombre, apellido, email)
VALUES ('pruebausuario_bajos_permisos', 'Prueba', 'User', 'prueba@pruebacorreo.com');
```

Pero sus permisos no lo permiten

```
3 13:28:09 INSERT INTO usuario (username, nombre, apellido, email) VAL... Error Code: 1142. INSERT command denied to user 'usuario_app'@'localhost' for table 'usuario'
```

Error Code: 1142. INSERT command denied to user 'usuario\_app'@'localhost' for table 'usuario'

Tampoco es posible actualizar

```
-- no es posible actualizar con este usuario ningun dato
UPDATE usuario SET estado='INACTIVO' WHERE id_usuario=1;
```

```
4 13:31:38 UPDATE usuario SET estado='INACTIVO' WHERE id_usuario=1 Error Code: 1142. UPDATE command denied to user 'usuario_app'@'localhost' for table 'usuario'
```

Error Code: 1142. UPDATE command denied to user 'usuario\_app'@'localhost' for table 'usuario'

### Ni borrar datos

```
-- Tampoco borrar
DELETE FROM usuario WHERE id_usuario=1;
```

```
Error Code: 1142. DELETE command denied to user 'usuario_app'@'localhost' for table 'usuario'
```

Error Code: 1142. DELETE command denied to user 'usuario\_app'@'localhost' for table 'usuario'



## 2. Pruebas de integridad

Ver 06\_seguridad\_vistas\_seguras.sql

Se ejecutan dos pruebas que demuestran el cumplimiento de las restricciones UNIQUE y FOREIGN KEY definidas en el esquema.

**Prueba 1:** Violación de la restricción UNIQUE (Duplicación de username)

Restricción probada: uq\_usuario\_username en la tabla usuario.

**Objetivo:** Demostrar que no se pueden insertar dos usuarios con el mismo username.

**Código para ingresar un usuario de prueba:**

```
INSERT INTO usuario (username, nombre, apellido, email) VALUES  
( 'usuario_duplicado', 'Juan', 'Perez', 'dup1@test.com' );
```

**Se vuelve a intentar nuevamente, ingresar un nuevo usuario con los mismos valores:**

```
33 -- se inserta un usuario de prueba  
34 • INSERT INTO usuario (username, nombre, apellido, email) VALUES ('usuario_duplicado', 'Juan', 'Perez', 'dup1@test.com');  
35  
36 -- intento de violacion de la regla  
37 • INSERT INTO usuario (username, nombre, apellido, email) VALUES ('usuario_duplicado', 'Pedro', 'Gomez', 'dup2@test.com');  
38  
39 /*
```

```
Error Code: 1062. Duplicate entry 'usuario_duplicado' for key 'usuario.uq_usuario_username'
```

Mensaje de Error Obtenido:

```
ERROR 1062 (23000): Duplicate entry 'usuario_duplicado' for key  
'gestion_usuarios.uq_usuario_username'
```

Conclusión: El motor de la base de datos bloquea la segunda inserción, garantizando que el campo username sea único.

## Prueba 2: Violación de la restricción de Integridad Referencial (Foreign Key - FK)

Restricción probada: fk\_credencial\_usuario en la tabla credencial\_acceso.

**Objetivo:** Demostrar que no se puede insertar una credencial para un usuario\_id que no existe en la tabla usuario.

Paso	Comando SQL	Resultado
Intento de Violación	<pre>INSERT INTO credencial_acceso (usuario_id, estado, hash_password, salt) VALUES (999999, 'ACTIVO', 'hash_falso', 'salt_falso');</pre>	<b>ERROR</b>

-- Prueba 2

```
INSERT INTO credencial_acceso (usuario_id, estado, hash_password, salt) VALUES (999999, 'ACTIVO', 'hash_falso', 'salt_falso');
```

El usuario con ID = 999999 no existe en nuestra base de datos

Mensaje de Error Obtenido:

```
Error Code: 1452. Cannot add or update a child row: a foreign key constraint fails (`gestion_usuarios_final`.`credencial_acceso`, CONSTRAINT `fk_credencial_usuario` FOREIGN KEY (`usuario_id`) REFERENCES `usuario` (`id_usuario`) ON DELETE CASCADE ON UPDATE CASCADE)
```

```
Error Code: 1452. Cannot add or update a child row: a foreign key constraint fails (`gestion_usuarios_final`.`credencial_acceso`, CONSTRAINT `fk_credencial_usuario` I
```

Conclusión: El sistema impide la creación de credenciales huérfanas, manteniendo la integridad referencial entre las tablas.

### 3. Procedimiento SQL seguro (sin SQL dinámico).

Ver 06\_seguridad\_vistas\_seguras.sql

Se elige la Opción B: Procedimiento Almacenado SQL sin SQL Dinámico. Este procedimiento se utiliza para actualizar la contraseña (hash y salt) de un usuario, garantizando que los datos de entrada se manejen como parámetros y no como código ejecutable.

```
DELIMITER //
```

```
CREATE PROCEDURE sp_actualizar_password_seguro (
```

```
    IN p_usuario_id INT,
```

```
    IN p_nuevo_hash VARCHAR(255),
```

```
    IN p_nuevo_salt VARCHAR(64)
```

```
)
```

```
BEGIN
```

```
    -- Utiliza parámetros de entrada que son tratados como datos.
```

```
    -- La consulta está predefinida y no se construye con  
concatenación de strings (NO ES SQL DINÁMICO).
```

```
    UPDATE credencial_acceso
```

```
    SET
```

```
    hash_password = p_nuevo_hash,
```

```
    salt = p_nuevo_salt,
```

```
    ultimo_cambio = NOW(),
```

```
    requiere_reset = FALSE
```

WHERE

usuario\_id = p\_usuario\_id;

END //

DELIMITER ;







-- Limpiamos el delimitador

-- Ejemplo de uso (Ejecución segura):

```
CALL sp_actualizar_password_seguro(1, 'nuevo_hash_cifrado_256',  
'nuevo_salt_32_chars');
```

Podemos ver su funcionamiento aqui:

```
341 -- Ejemplo de uso (Ejecución segura):  
342 CALL sp_actualizar_password_seguro(1, 'nuevo_hash_cifrado_256', 'nuevo_salt_32_chars');  
343  
344 • select * from credencial_acceso where id_credencial = 1;  
345
```

Result Grid									
Filter Rows: <input type="text"/>									
Edit:   									
Export/Import:  									
Wrap Cell Content: 									
	id_credencial	eliminado	usuario_id	estado	ultima_sesion	hash_password	salt	ultimo_cambio	requiere_reset
▶	1	0	1	inactivo	2025-10-21 00:00:01	nuevo_hash_cifrado_256	nuevo_salt_32_chars	2025-10-22 14:11:27	0
*	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL

hash_password	salt	ultimo_cambio
nuevo_hash_cifrado_256	nuevo_salt_32_chars	2025-10-22 14:11:27
NULL	NULL	NULL

## 4. Prueba anti-inyección documentada

### ver 06\_seguridad\_vistas\_seguras.sql

Se simula un ataque de inyección SQL para demostrar la seguridad del procedimiento almacenado implementado.

Opción Elegida: Procedimiento Almacenado SQL sin SQL Dinámico  
(sp\_actualizar\_password\_seguro).

- **Entrada Maliciosa Utilizada**

Se intenta inyectar código malicioso en el parámetro p\_nuevo\_hash para borrar una tabla.

### Comando de Ejecución (Intento Malicioso)

```
CALL sp_actualizar_password_seguro(1, 'hash_falso'; DROP  
TABLE credencial_acceso; --', 'un_salt');
```

- **Resultado**

La tabla credencial\_acceso NO se elimina. El registro del usuario\_id = 1 es actualizado, y el valor del hash\_password contendrá la cadena completa inyectada.

- **Resultado Observado**

```
SELECT hash_password FROM credencial_acceso WHERE usuario_id =  
1; muestra: 'hash_falso'; DROP TABLE credencial_acceso; --'.  
No hay error de sintaxis y la tabla permanece intacta.
```

- **Explicación de por qué quedó neutralizada**

El procedimiento almacenado es inherentemente seguro contra la mayoría de los ataques de Inyección SQL porque **utiliza parámetros**. El motor de la base de datos trata la cadena inyectada ('hash\_falso'; DROP TABLE credencial\_acceso; --') **como**

un simple dato de texto para ser almacenado en el campo hash\_password, y nunca lo interpreta como una nueva instrucción SQL (DROP TABLE). Esto previene que el código malicioso se ejecute.

### Intento de inyección malicioso:

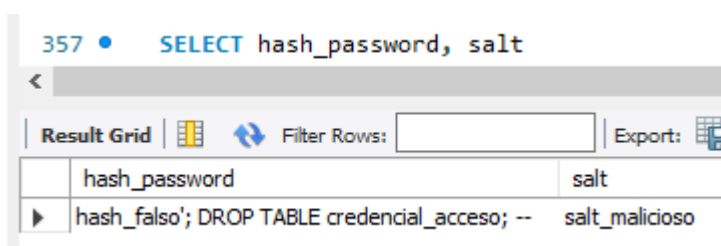
```
-- intento de inyeccion maliciosa, quiere borrar la tabla de credencial de acceso
CALL sp_actualizar_password_seguro(
    1,
    'hash_falso'; DROP TABLE credencial_acceso; --',
    'salt_malicioso'
);
```

Este código contiene un DROP TABLE entre su código, intentado que el compilador lo lea como:

```
SET hash_password = 'hash_falso';
DROP TABLE credencial_acceso; --',
salt = 'salt_malicioso'
WHERE usuario_id = 1;
```

### Así con drop table eliminando toda la tabla

Utilizando el procedimiento seguro, la tabla continúa solo con los cambios que se ingresaron de hash



357 • SELECT hash\_password, salt

hash_password	salt
hash_falso'; DROP TABLE credencial_acceso; --'	salt_malicioso

Pero es posible verificar que la tabla continua existiendo con sus datos:

id_credencial	eliminado	usuario_id	estado	ultima_sesion	hash_password	salt
7	0	8	ACTIVO	2025-10-14 00:00:08	64B489764C31ABC1C1D7855DCD1F0235C532...	B66439E602D61438BDB7D1BAEAE2237C
8	0	9	ACTIVO	2025-10-13 00:00:09	2ADDDBC8D6E469E7423E844A7BC215297DE0...	A9B75FEE5CC375951EE730CA6355872C
9	0	11	ACTIVO	2025-10-11 00:00:11	75E2E883B82F3F95D951C41F95B0ABC8E93CA...	7A045665191D6C0C5CB2ECE5AFD97A5C
10	0	12	ACTIVO	2025-10-10 00:00:12	8C6F72B903DB70F14403459588C9DA9929529...	3CF5D7097F418DBE58036E91D3C493F5
11	0	13	ACTIVO	2025-10-09 00:00:13	161CA71F4D4EB933DCBE2CC63DCC631660FA...	2634F7628C7D1372F7F6691742FA7161
12	0	14	ACTIVO	2025-10-08 00:00:14	8B58EFA30B678D1DD1A578CAB74FA37D25B6...	9A62C496E4F9DF8804437186741FF34F
13	0	15	INACTIVO	2025-10-07 00:00:15	3F7B3E101BB98D0F8851ECD6774ABF412CEA3...	DD40792E58D49A5A11BF6FB8E2A1EDFF
14	0	16	ACTIVO	2025-10-06 00:00:16	A5BD72E5834ED63E1FC2F9702F8544B0EF92E...	2104410F4810A5BFC5FEFC444F95A6A
15	0	17	ACTIVO	2025-10-05 00:00:17	F3462DAD009DC109A380F9F8A605D942E37C...	BAC95DBACBE85F3FA095BDC3445D2996
16	0	19	ACTIVO	2025-10-03 00:00:19	4D95C4E288BD85DDB4F741594367545167E47...	AEABDF9D08156AC859B9E6A3B8FBF174
17	0	20	INACTIVO	2025-10-02 00:00:20	9BD37D07F461DE723991D7B19B8346D3445F4...	25DA2B508D6E4CC5B67F0D16B45C673A
18	0	21	ACTIVO	2025-10-01 00:00:21	A91D57262A42C48DEEDA78A2FE93605143087...	9B97E7F7FD175C68E42D4EEB1A535559
19	0	22	ACTIVO	2025-09-30 00:00:22	8590183CBA650139502658C770EB5572C3C33...	144A1CD23B1E507B6918991A0A787496
20	0	23	ACTIVO	2025-09-29 00:00:23	60FFF97304749C643E7E9027D504B39F2D9A2...	D95011E06E1A41A9CF72347CC4F2E9A7
21	0	24	ACTIVO	2025-09-28 00:00:24	155A309EC19E29ABE3F3E69582BB27B3033BC...	7FC907BD34306779B6D4190A10AC298D
22	0	25	INACTIVO	2025-09-27 00:00:25	844066F8764A633AE67F469F292DFFACEDF17...	3DB70B4CBCE9819EA240E5D0926D87F6
23	0	27	ACTIVO	2025-09-25 00:00:27	1A284497B8F697CBD1869B07DF4EC33220915...	8C3C47B9FB62ED46E5C9F2DF6B4FFFE4
24	0	28	ACTIVO	2025-09-24 00:00:28	DFB1816374583324C93191FE0D0A4298A1740...	354671D59F3AB7F806DE611516911489
25	0	29	ACTIVO	2025-09-23 00:00:29	A6775D4759F635947B176A18E678D92F39534...	58FCF32DEA0602CF80579D132C471E32
26	0	30	INACTIVO	2025-09-22 00:00:30	C49C70868BBEDDF39BF44B58E51133AE4BA16...	A6694BB9E806D8CBA8682B53493D8286
27	0	31	ACTIVO	2025-09-21 00:00:31	B10DBD46B41DBCACB58843B8ECCD3E9C18BB...	A07E8735F624E3A79DEF116B7FE17C76
28	0	32	ACTIVO	2025-09-20 00:00:32	9B48B54D84DD8A453B3CD43DBD478A735E8...	69827D1362F50569CF20286FAD1A8C57
29	0	33	ACTIVO	2025-09-19 00:00:33	EFF6372DB51355698B35F86E768D6E0D7D99D...	3D5B996413262EB401FED0F564DA067E
30	0	35	INACTIVO	2025-09-17 00:00:35	7B47432126447CC6403C1645AD1EAC321536B...	09A4AF230306C20AD84D8A52781D862F

## **Etapla 5. Concurrency y transacciones**

Ver 08\_Concurrency\_guiada.sql  
y 08\_Concurrency\_guiada\_sesion2.sql

### **1. Simulación de Deadlocks**

Comenzamos una transacción para simular un deadlock (en este caso es actualizar el valor de activo del mismo usuario)

#### **1. En la sesión 1:**

```
START TRANSACTION;  
  
UPDATE usuario  
  
    SET activo = FALSE  
  
    WHERE id_usuario = 1;  
  
-- Esto Bloquea la fila con id_usuario = 1
```

#### **2. En sesión 2:**

```
START TRANSACTION;  
  
UPDATE usuario  
  
    SET activo = FALSE  
  
    WHERE id_usuario = 2;  
  
-- bloquea la fila con id_usuario = 2
```



### **3. Volvemos a la sesion 1:**

```
UPDATE usuario
  SET activo = TRUE
  WHERE id_usuario = 2;
-- Queda esperando. Sesión 2 ya tiene lock sobre id_usuario = 2
```

### **4. Finalmente en la sesion 2:**

```
UPDATE usuario

  SET activo = TRUE

  WHERE id_usuario = 1;

-- Aquí MySQL detecta el ciclo de espera y lanza un DEADLOCK

Error:

Error Code: 1213. Deadlock found when trying to get lock; try restarting transaction
```

## Sesión 1:

```
-- ===== simulando deadlock con usuario activo =====

-- paso 1 sesion 1:
START TRANSACTION;
UPDATE usuario
  SET activo = FALSE
 WHERE id_usuario = 1;
-- Bloquea la fila con id_usuario = 1

-- paso 3 sesion 1
UPDATE usuario
  SET activo = TRUE
 WHERE id_usuario = 2;
-- Queda esperando porque Sesión 2 ya tiene lock sobre id_usuario = 2
```

## Sesión 2:

```
-- ===== simulando deadlock con usuario activo =====

-- paso 2 sesion 2:

START TRANSACTION;
UPDATE usuario
  SET activo = FALSE
 WHERE id_usuario = 2;
-- bloquea la fila con id_usuario = 2

-- paso 4 sesion 2
UPDATE usuario
  SET activo = TRUE
 WHERE id_usuario = 1;
-- Aquí MySQL detecta el ciclo de espera y lanza un DEADLOCK
```

## Error de deadlock:

✖	46	16:56:22	UPDATE usuario SET activo = TRUE WHERE id_usuario = 1
---	----	----------	---

Error Code: 1213. Deadlock found when trying to get lock; try restarting transaction

## Como solucionar este deadlock:

Utilizando un bloqueo

### 1. Desde la sesion 1:

(Ejecutar sin el commit para probar el bloqueo en la segunda sesion)

```
START TRANSACTION;  
SELECT * FROM usuario WHERE id_usuario IN (1,2) FOR UPDATE;
```

```
UPDATE usuario  
    SET activo = FALSE  
    WHERE id_usuario = 1;
```

```
UPDATE usuario  
    SET activo = FALSE  
    WHERE id_usuario = 2;  
COMMIT;
```

Antes de realizar el commit podemos ver los cambios, pero las tablas continuan bloqueadas

	id_usuario	eliminado	username	nombre	apellido	email	fecha_registro	activo
▶	1	0	UserTest1	Ana	Gómez	mailtest1@example.com	2023-12-17 18:55:49	0
	2	0	UserTest2	Juan	Perez	mailtest2@example.com	2023-12-07 18:55:49	0
*	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL

## 2. Desde la sesion 2

```
START TRANSACTION;  
SELECT * FROM usuario WHERE id_usuario IN (1,2) FOR UPDATE;
```

```
UPDATE usuario  
  SET activo = TRUE  
  WHERE id_usuario = 1;
```

```
UPDATE usuario  
  SET activo = TRUE  
  WHERE id_usuario = 2;
```

```
COMMIT;
```

Se puede ver como la sesion 2 espera que se genere el commit de la sesion 1, que tiene la tabla bloqueada:

4	11	19:18:37	SELECT * FROM usuario WHERE id_usuario IN (1,2) LIMIT 0, 1000 FOR UPDATE	Running...
---	----	----------	--	------------

Al realizar el commit desde la sesion 1, la sesion 2 puede terminar:

13	19:19:00	UPDATE usuario SET activo = TRUE WHERE id_usuario = 2	0 row(s) affected Rows matched: 1 Changed: 0 Warnings: 0
----	----------	---	--

**Al final, los cambios de la sesion 2 se aplican. Cuando la primera libera el bloqueo**

	id_usuario	eliminado	username	nombre	apellido	email	fecha_registro	activo
▶	1	0	UserTest1	Ana	Gómez	mailtest1@example.com	2023-12-17 18:55:49	1
	2	0	UserTest2	Juan	Perez	mailtest2@example.com	2023-12-07 18:55:49	1
*	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL

## 2. Procedimiento almacenado en SQL

**Ver 07\_transacciones.sql**

**Primero se crea la tabla para almacenar los errores en caso que ocurran:**

```
-- Tabla para registrar los errores, incluyendo deadlocks
CREATE TABLE log_errores (
    id_log INT NOT NULL AUTO_INCREMENT PRIMARY KEY,
    fecha_hora DATETIME NOT NULL DEFAULT CURRENT_TIMESTAMP,
    procedimiento VARCHAR(100) NOT NULL,
    nivel VARCHAR(10) NOT NULL, -- 'ERROR', 'WARN', 'INFO'
    codigo_sql INT NULL,
    estado_sql VARCHAR(5) NULL,
    mensaje_error TEXT NOT NULL,
    intentos INT NULL
) ENGINE=InnoDB;
```

**Se crea un procedimiento para cambiar si un usuario esta activo**

```
DELIMITER $$
-- delimiter cambia el punto y coma por $$ para poder utilizar ;
dentro de la funcion

DROP PROCEDURE IF EXISTS p_actualizar_estado_activo_usuario $$
CREATE PROCEDURE p_actualizar_estado_activo_usuario(
    IN p_id_usuario INT,
    IN p_nuevo_activo BOOLEAN
    -- identifica la fila del usuario y el valor que vamos a cambiar
)
BEGIN
    -- control de reintentos y estado
    DECLARE v_max_intentos INT DEFAULT 2;
    DECLARE v_intento_actual INT DEFAULT 0;
    DECLARE v_exito BOOLEAN DEFAULT FALSE;
    DECLARE v_deadlock_detectado BOOLEAN DEFAULT FALSE;
    -- crea las variables necesarias
```

```

-- datos para logging
DECLARE v_codigo_error INT;
DECLARE v_estado_sql VARCHAR(5);
DECLARE v_mensaje_error TEXT;

-- handler para deadlock (SQLSTATE 40001) (es el error de
deadlock)
DECLARE CONTINUE HANDLER FOR SQLSTATE '40001'
BEGIN
    SET v_deadlock_detectado = TRUE;
    ROLLBACK;
    SET v_codigo_error = 1213;
    SET v_estado_sql = '40001';
    SET v_mensaje_error = CONCAT(
        'Deadlock al cambiar activo. Intento ', v_intento_actual
+ 1
    );
    INSERT INTO log_errores (
        procedimiento, nivel, codigo_sql, estado_sql,
mensaje_error, intentos
    ) VALUES (
        'p_actualizar_estado_activo_usuario',
        'WARN',
        v_codigo_error,
        v_estado_sql,
        v_mensaje_error,
        v_intento_actual + 1
    );
    SELECT SLEEP(0.1);
END;

-- en caso de encontrar un deadlock, hace un rollback,
captura los detalles y los almacena en la tabla de errores

-- bucle de retry
WHILE v_intento_actual <= v_max_intentos
    AND v_exito = FALSE
DO
    SET v_intento_actual = v_intento_actual + 1;

```

```

SET v_deadlock_detectado = FALSE;

START TRANSACTION;
    UPDATE usuario
    SET activo = p_nuevo_activo
    WHERE id_usuario = p_id_usuario;
    -- si no hubo deadlock, confirmamos y marcamos éxito
    IF v_deadlock_detectado = FALSE THEN
        COMMIT;
        SET v_exito = TRUE;
    END IF;
END WHILE;
    -- este bucle reintenta la transaccion hasta que alcanza el
    maximo de v_max_intentos

    -- si tras todos los intentos no hubo éxito
    IF v_exito = FALSE THEN
        IF v_deadlock_detectado = TRUE THEN
            -- si no tuvo existo y ademas encontro un deadlock:
            -- guarda en log_errores el error
            INSERT INTO log_errores (
                procedimiento, nivel, codigo_sql, estado_sql,
mensaje_error, intentos
            ) VALUES (
                'p_actualizar_estado_activo_usuario',
                'ERROR',
                1213,
                '40001',
                CONCAT(
                    'FALLO DEFINITIVO: No se pudo cambiar activo de
usuario ',
                    p_id_usuario,
                    ' tras ',
                    v_intento_actual,
                    ' intentos'
                ),
                v_intento_actual
            );

```

```

        SIGNAL SQLSTATE '45000'
            SET MESSAGE_TEXT = 'Error permanente por deadlocks.
Intente más tarde.',
            MYSQL_ERRNO = 50005;
    ELSE
        -- rollback de seguridad si quedara activa
        IF EXISTS (
            SELECT 1
            FROM information_schema.innodb_trx
            WHERE trx_mysql_thread_id = CONNECTION_ID()
        ) THEN
            ROLLBACK;
        END IF;
        SIGNAL SQLSTATE '45000'
            SET MESSAGE_TEXT = 'Error inesperado al cambiar
activo.',
            MYSQL_ERRNO = 50006;
    END IF;
END IF;
END $$
-- cierra el cuerpo del procedimiento con (como si fuera end;)
DELIMITER ;
-- vuelve a cambiar el delimiter a ;

```

#### Probando p\_actualizar\_estado\_activo\_usuario :

```

CALL p_actualizar_estado_activo_usuario( 1, TRUE );
SELECT *
FROM usuario
WHERE id_usuario = 1;

```

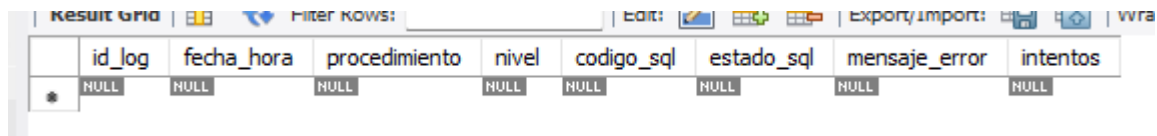
Result Grid   Filter Rows:   Edit:   Export/Import:   Wrap Cell Center								
	id_usuario	eliminado	username	nombre	apellido	email	fecha_registro	activo
▶	1	0	Test1	Matias	Carro	Carro_Test1@mail.com	2025-10-18 13:46:47	1
*	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL

Se puede confirmar que cambio el estado a 1



**Se verifica el log de errores:**

```
SELECT *  
  
FROM log_errores  
  
WHERE procedimiento = 'p_actualizar_estado_activo_usuario'  
  
ORDER BY id_log DESC  
  
LIMIT 5;
```



The screenshot shows a 'Result Grid' window from a database client. It contains a single row of data with the following columns: id\_log, fecha\_hora, procedimiento, nivel, codigo\_sql, estado\_sql, mensaje\_error, and intentos. All values in this row are NULL.

id_log	fecha_hora	procedimiento	nivel	codigo_sql	estado_sql	mensaje_error	intentos
NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL

No se crea ningún error ya que la transacción fue completada correctamente.

### 3 . READ COMMITTED y REPEATABLE READ

Se realizan dos tipos de transacciones usando los dos métodos y dos sesiones distintas para las pruebas (conecciones en workbench)

#### READ COMMITTED

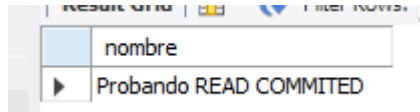
**1. Se modifica al usuario con id 1 para la prueba**

```
UPDATE usuario  
SET nombre = 'Probando READ COMMITED'  
WHERE id_usuario = 1;  
COMMIT;
```

**2. Se comienza la transaccion con Read Committed:**

```
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;  
START TRANSACTION;  
SELECT nombre  
FROM usuario  
WHERE id_usuario = 1;
```

**Resultado:**



nombre
Probando READ COMMITED

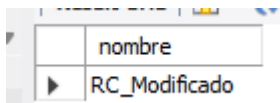
**3. Se modifica el nombre del usuario desde la sesion 2**

```
UPDATE usuario  
  SET nombre = 'RC_Modificado'  
  WHERE id_usuario = 1;  
COMMIT;
```

**4. Se vuelve a la sesion 1 y se solicita el nombre del usuario**

```
SELECT nombre  
  FROM usuario  
  WHERE id_usuario = 1;  
ROLLBACK;
```

**Resultado:**



nombre
RC_Modificado

Se puede ver como se modifico el valor del usuario y lo realizado en la sesion 2 impacta en la sesion 1

## REPEATABLE READ

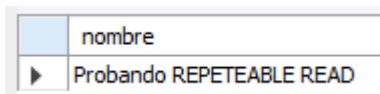
**1. Se ingresa un valor en el usuario con id 1 para mostrar la prueba**

```
UPDATE usuario  
  SET nombre = 'Probando Repeteable Read'  
  WHERE id_usuario = 1;  
COMMIT;
```

**2. Aun en la sesion 1, se inicia la transacción**

```
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;  
START TRANSACTION;  
  SELECT nombre  
    FROM usuario  
   WHERE id_usuario = 1;
```

**Resultado:**

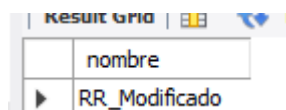


nombre
Probando REPETEABLE READ

**3. Desde la sesion 2, se realizan modificaciones**

```
UPDATE usuario  
  SET nombre = 'RR_Modificado'  
 WHERE id_usuario = 1;  
COMMIT;  
  
SELECT nombre  
  FROM usuario  
 WHERE id_usuario = 1;
```

**Resultado:**



nombre
RR_Modificado

**4. Se vuelve a la sesion 1 donde se realiza una consulta.**

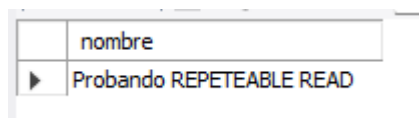
```
-- PASO final sesion 1  
SELECT nombre
```

```

        FROM usuario
        WHERE id_usuario = 1;
-- Resultado: 'Original' porque REPEATABLE READ mantiene la
snapshot de la primera lectura
ROLLBACK;
-- cierra la transacción y no guarda los cambios realizados

```

### Resultado:



nombre
Probando REPEATABLE READ

Se puede ver como Repeatable Read mantuvo para visualizar la información que se creó en la transacción de la sesión 1.

## Impacto de índices en entornos concurrentes

### Ver 08\_Concurrencia\_guiada.sql

Se realiza prueba para mostrar el impacto de los índices:  
Utilizando EXPLAIN ANALYZE para poder ver el tiempo que lleva realizar un select en una transacción.

```

START TRANSACTION;
EXPLAIN ANALYZE
SELECT * FROM usuario WHERE apellido = 'Castro' FOR UPDATE;
COMMIT;

```

```

START TRANSACTION;
EXPLAIN ANALYZE
SELECT * FROM usuario WHERE apellido = 'Castro' FOR UPDATE;
-- no aplica commit, iniciar el mismo en la conexion 2
COMMIT;

```

Siendo el resultado:

```

-> Filter: (usuario.apellido = 'Castro') (cost=10282 rows=10098)
(actual time=15.7..43.6 rows=5000 loops=1)      -> Table scan on
usuario      (cost=10282 rows=100981) (actual time=0.0569..38.2
rows=100000 loops=1)

```

Se confirma que hace un scan completo y el tiempo que lleva.

Para poder simular un uso muy grande o una transacción que bloquea la tabla mientras lee, se corre la transacción sin utilizar EXPLAIN ANALYZE Y COMMIT:

La primer sesion nos da un retorno:

	id_usuario	eliminado	username	nombre	apellido	email	fecha_registro	activo	estado
▶	36011	0	sofia.castro.9030	Sofia	Castro	sofia.castro.9030@example.com	2025-01-25 09:49:36	0	INACTIVO
	36012	0	sofia.castro.19030	Sofia	Castro	sofia.castro.19030@example.com	2025-09-02 09:49:36	0	ACTIVO
	36013	0	sofia.castro.29030	Sofia	Castro	sofia.castro.29030@example.com	2024-04-10 09:49:36	0	ACTIVO
	36014	0	sofia.castro.39030	Sofia	Castro	sofia.castro.39030@example.com	2024-11-16 09:49:36	0	ACTIVO
	36015	0	sofia.castro.49030	Sofia	Castro	sofia.castro.49030@example.com	2025-06-24 09:49:36	0	ACTIVO

Mientras la segunda sesion queda en espera, realizando otra busqueda:

```

START TRANSACTION;
-- EXPLAIN ANALYZE
SELECT * FROM usuario WHERE apellido = 'Silva' FOR UPDATE;
-- esperar no realizar commit
COMMIT;

```

4	28	11:26:37	SELECT * FROM usuario WHERE apellido = 'Castro' LIMIT 0, 1000 FOR UPDATE	Running...
---	----	----------	--	------------

Luego de hacer commit en la primera podemos ver todo el tiempo que tarda en hacer esa búsqueda mientras la sesión 1 trabaja:

✓	30	11:27:27	SELECT * FROM usuario WHERE apellido = 'Castro' LIMIT 0, 1000 FOR UPDATE	4.125 sec / 0.000 sec	1000 row(s) returned
---	----	----------	--	-----------------------	----------------------

## Creando índice para prueba:

```
CREATE INDEX ix_usuario_apellido ON usuario (apellido);
```

Se crea un índice para los apellidos de los usuarios y se pasa a realizar más pruebas:

Se vuelve a realizar la búsqueda de todos los usuarios con apellido “Castro” desde la sesión 1:

```
START TRANSACTION;
-- EXPLAIN ANALYZE
SELECT * FROM usuario WHERE apellido = 'Castro' FOR UPDATE;
-- no aplica commit, iniciar el mismo en la conexión 2
COMMIT;
```

No se realiza commit, para simular que la transacción lleva tiempo y está siendo utilizada la tabla.

Luego, desde la sesión 2, se busca otro apellido: “Silva”

```
START TRANSACTION;
-- EXPLAIN ANALYZE
SELECT * FROM usuario WHERE apellido = 'Silva' FOR UPDATE;
-- esperar no realizar commit
COMMIT;
```

Ambas  
sesiones  
devuelven  
la

información de manera rápida, a pesar de que esta siendo “bloqueada” por la sesion 1:

### Sesión 1:

✓	342	11:33:30	SELECT * FROM usuario WHERE apellido = 'Castro' LIMIT 0, 1000 FOR UPDATE	0.015 sec / 0.000 sec	1000 row(s) returned
---	-----	----------	--	-----------------------	----------------------

### Sesión 2:

✓	36	11:33:35	SELECT * FROM usuario WHERE apellido = 'Silva' LIMIT 0, 1000 FOR UPDATE	0.015 sec / 0.000 sec	1000 row(s) returned
---	----	----------	---	-----------------------	----------------------

Incluso mientras la sesion 1 mantenía bloqueado por la transacción “no se uso commit” la sesion 2 pudo devolver la búsqueda rápidamente:

	id_usuario	eliminado	username	nombre	apellido	email	fecha_registro	activo	estado
▶	1	0	valentina.silva.90031	S1	Silva	valentina.silva.90031@example.com	2025-02-23 09:49:36	1	ACTIVO
	2	0	valentina.silva.9131	Valentina	Silva	valentina.silva.9131@example.com	2024-10-16 09:49:36	1	ACTIVO
	3	0	valentina.silva.19131	Valentina	Silva	valentina.silva.19131@example.com	2025-05-24 09:49:36	1	INACTIVO
	4	0	valentina.silva.29131	Valentina	Silva	valentina.silva.29131@example.com	2023-12-31 09:49:36	1	ACTIVO
	5	0	valentina.silva.39131	Valentina	Silva	valentina.silva.39131@example.com	2024-08-07 09:49:36	1	ACTIVO

Debido a que se trata de filas distintas (otro apellido) y esta utilizando el indice:

**EXPLAIN ANALYZE**

**SELECT \* FROM usuario WHERE apellido = 'Silva' FOR UPDATE;**

-> Index lookup on usuario using ix\_usuario\_apellido (apellido='Silva') (cost=1053 rows=5000) (actual time=0.13..6.5 rows=5000 loops=1)

EXPLAIN
▶ -> Index lookup on usuario using ix_usuario_apellido (apellido='Silva') (cost=1053 rows=5000) (actual time=0.13..6.5 rows=5000 loops=1)

Por lo tanto, la búsqueda en la sesión 1 solo bloquea las filas que ya tiene indexadas y aunque se encuentre en espera de commit, la sesión 2 puede trabajar en otras filas diferentes gracias al índice.

## Informe Etapa 5

En la práctica se observó que el manejo de transacciones es clave para garantizar la consistencia de los datos en entornos multiusuario.

Bajo READ COMMITTED, cada consulta accede a la versión más reciente confirmada, lo que permite lecturas distintas dentro de la misma transacción (lecturas no repetibles). En cambio, con REPEATABLE READ se mantiene un “snapshot” estable desde el inicio de la transacción, asegurando que las lecturas sucesivas devuelvan siempre los mismos resultados, aunque otras sesiones hayan modificado los datos. Esto muestra cómo el nivel de aislamiento define el equilibrio entre concurrencia y consistencia: niveles más bajos permiten mayor concurrencia pero menos estabilidad en las lecturas, mientras que niveles más altos reducen anomalías a costa de mayor bloqueo o espera. El uso de COMMIT y ROLLBACK es esencial para confirmar o deshacer cambios y liberar recursos, evitando inconsistencias y bloqueos prolongados.



# Anexo 1: Uso de la IA

## Etapas 1 – Modelado y Definición de Constraints:

### Prompt:

“Estoy desarrollando un script SQL para generar 100.000 registros ficticios en la tabla ventas. Estoy trabajando en MySQL. Necesito que me guíes paso a paso, sin darme todo el código de golpe. voy pasarte la primer instancia de la creacion de la base de datos. donde ya diseñe parte de las tablas:

```
id_usuario VARCHAR(20) PRIMARY KEY,  
username VARCHAR(50)  
nombre VARCHAR(100)  
apellido VARCHAR(100)  
email VARCHAR(100)  
fecha_registro DATE
```

```
id_credencial INT PRIMARY KEY,  
usuario_id VARCHAR(20)  
password_hash VARCHAR(255)  
estado VARCHAR(10)
```

la etapa 1 consiste en:

Modelado y Definición de Constraints: Elaborar el DER y el modelo relacional con PRIMARY KEY, FOREIGN KEY, UNIQUE, CHECK y restricciones de dominio.”

### Respuesta:

Objetivo de la etapa

Vas a transformar tu diseño lógico en un modelo relacional sólido, con claves y restricciones que protejan la integridad, preparándolo para escalar a grandes volúmenes (como tus 100.000 ventas). Pensá esto como “poner rejas y reglas” en la base: que solo entren datos válidos, únicos y consistentes.

Der y relaciones clave

- **Entidad usuario:** representa la identidad base de la persona.
- **Entidad credencial:** representa el acceso/autenticación, 1:1 con usuario.
- **Relación 1:1:** cada usuario tiene una sola credencial, y cada credencial pertenece a un único usuario.
- **Clave foránea:** en credencial, apuntando a la PK de usuario.

- **Cardinalidad y obligatoriedad:** decidí si la credencial es obligatoria desde el alta o puede crearse después; eso define si la FK permite NULL y la política de creación.

#### Definición de claves y unicidad

- **Primary keys:**
  - **Usuario:** evaluá si usar tu clave natural `id_usuario` (VARCHAR) o una clave sustituta (INT autoincremental/UUID).
    - **Regla:** con claves naturales, definí claramente el dominio y la inmutabilidad. Con claves sustitutas, simplificás FKs, índices y joins.
  - **Credencial:** si mantenés `id_credencial` como PK, asegurate que sea estable y eficiente para joins.
- **Unicidad funcional:**
  - **Username:** debe ser único; es parte de la identidad pública.
  - **Email:** definí unicidad según tu negocio (único por persona o permitir duplicados en casos especiales). En la mayoría de sistemas, es único.
- **Combinaciones únicas opcionales:** si en algún momento necesitás enforcing de 1:1, asegurá que `usuario_id` sea único en credencial.

#### Restricciones de dominio y CHECK

- **Estado en credencial:** restringí a un conjunto cerrado (ej.: “activo”, “bloqueado”, “pendiente”).
- **Password hash:** establecé longitud y conjunto de caracteres esperados según el algoritmo (ej.: hex/base64). No guardes contraseñas en claro.
- **Email:** validá formato básico en base de datos (CHECK simple), y hacé validación profunda en aplicación.
- **Fechas:** `fecha_registro` no debe ser futura si tu modelo lo requiere; decidí si permitís valores por defecto automáticos.
- **Nombre/apellido:** definí si admiten sólo letras y espacios (esto suele ser validación de aplicación; en BD podés limitar longitud y no null).

#### Nullabilidad y reglas de negocio

- **Campos obligatorios:** `username`, `nombre`, `apellido`, `email` y `fecha de registro` deberían ser NOT NULL si son requeridos al alta.
- **Campos opcionales:** definí claramente qué puede quedar NULL (ej.: algún estado transicional).
- **Credencial obligatoria:** si el acceso es parte del alta, `usuario_id` en credencial debe ser NOT NULL y con unicidad 1:1; si no, permití credenciales posteriores.

#### Integridad referencial y cascadas

- **FK credencial → usuario:** asegura que no existan credenciales huérfanas.
- **Política de borrado:**

- **Baja lógica:** si usás eliminado en usuario, no deberías cascadar borrados físicos; mantené consistencia con flags.
- **Borrado físico:** si alguna vez se borra usuario definitivamente, definí si la credencial se borra en cascada.
- **Actualización de claves:** evitá actualizar PKs naturales; si existe esa posibilidad, definí la política de actualización de la FK.

#### Índices y desempeño

- **Índices en columnas de búsqueda:** username y email deben tener índices únicos (ya lo tenés); considerá índices en usuario\_id en credencial para joins rápidos.
- **Índices por consulta frecuente:** si vas a filtrar por estado o por fecha\_registro, evaluá índices complementarios.
- **Trade-offs:** más índices aceleran lecturas pero encarecen escrituras; ajustá al patrón de acceso real.

#### Normalización y consistencia

- **Tercera Forma Normal:** separaste identidad (usuario) de credenciales (acceso), lo cual evita redundancia y fugas de seguridad.
- **Evitar derivadas almacenadas:** no dupliques información que pueda recalcularse; registra eventos como “última sesión” en credencial si lo usás para auditoría, pero evita campos inconsistentes.

#### Seguridad y buenas prácticas

- **Separación de responsabilidades:** no mezcles datos personales con secretos; el hash va en credencial, nunca guardes contraseñas.
- **Control de acceso:** planeá vistas que oculten datos sensibles (hash, salt) y da permisos SELECT sobre vistas, no sobre tablas crudas.
- **Validación dual:** validá en la aplicación y reforzá en la base con restricciones declarativas.

## Reglas de negocios

### prompt

“Que serian las reglas de negocio? como se aplican a este caso?”

### Respuesta:

Muy buena pregunta.

Las **reglas de negocio** son las condiciones, restricciones o políticas que definen **cómo debe funcionar el sistema** de acuerdo con la lógica de la organización. No son reglas técnicas de SQL en sí mismas, sino **normas que reflejan la realidad del negocio** y que luego se traducen en constraints, triggers, validaciones o lógica de aplicación.

En este caso (usuarios y credenciales de acceso), ejemplos de reglas de negocio:

- **Un usuario debe tener una credencial de acceso válida** (no puede existir credencial sin usuario asociado).
- **El nombre de usuario debe ser único** en todo el sistema.
- **La contraseña debe cumplir requisitos mínimos** (ejemplo: al menos 8 caracteres, incluir mayúsculas, minúsculas y números).
- **Un usuario puede tener solo una credencial activa a la vez** (si se permite histórico, las anteriores deben marcarse como inactivas).
- **El correo electrónico debe tener formato válido** y no repetirse entre usuarios.
- **La fecha de creación de la credencial no puede ser futura.**
- **Si un usuario está marcado como “bloqueado”, no puede iniciar sesión.**

### Prompt

La FK de credencial seria id\_usuario. que seria lo mejor al declarar en on update o en un delete? en update lo mejor seria actualizar las correspondientes no?

### Respuesta:

#### ¿Cuál es la FK?

- Sí, en tu tabla credencial\_acceso la **FK natural** es id\_usuario.
- Esa columna debe apuntar a la **PK** de la tabla usuario (id\_usuario INT AUTO\_INCREMENT).
- Para que la relación sea 1:1, además de la FK, conviene que id\_usuario sea **UNIQUE** en credencial\_acceso.

### ON UPDATE

En teoría, una relación puede configurarse para que, si cambia el valor de la clave primaria en la tabla padre, ese cambio se propague automáticamente a la tabla hija. Sin embargo, en la práctica, los identificadores autoincrementales cumplen el rol de **identificadores estables**: se generan una vez y no se modifican.

Por esa razón, la mayoría de los diseñadores no esperan que un identificador cambie y suelen dejar la acción de actualización restringida o sin efecto. Aunque técnicamente se podría permitir la actualización en cascada, casi nunca se utiliza porque no responde a un escenario realista.

### ON DELETE

Aquí sí aparece una verdadera decisión de diseño, vinculada a la **lógica de negocio** y a cómo se quiere manejar la integridad de los datos:

- **Cascada (CASCADE)**: si se elimina un registro en la tabla padre, automáticamente se eliminan los registros relacionados en la tabla hija. Esto asegura que no queden datos huérfanos.
- **Restricción o no acción (RESTRICT / NO ACTION)**: impide la eliminación en la tabla padre si existen registros dependientes en la hija. Protege la integridad, pero obliga a resolver manualmente esas dependencias antes de borrar.
- **Asignación a nulo (SET NULL)**: si se elimina el registro padre, el campo correspondiente en la tabla hija queda vacío. En relaciones uno a uno no suele ser lógico, porque dejaría un registro incompleto o sin sentido.

### Aplicación al caso de usuarios y credenciales

En un modelo donde cada usuario tiene asociada una credencial, lo más coherente es que, al eliminar al usuario, también desaparezca su credencial. Dejarla sin referencia carecería de sentido práctico. Por eso, en este tipo de escenarios, la opción más lógica es la eliminación en cascada.

---

## Etapa 2. Generación de datos masivos con SQL puro

Para la realización de este trabajo utilicé asistencia de inteligencia artificial como herramienta de apoyo técnico y pedagógico. La IA me ayudó principalmente a verificar la sintaxis y estructura de las sentencias SQL, asegurando la correcta implementación de claves primarias, foráneas, restricciones de unicidad y validaciones CHECK.

Durante el desarrollo, también le consulté cuál sería la mejor forma de realizar una carga masiva de datos, evaluando la posibilidad de hacerlo mediante un proceso de tipo BULK INSERT o generando los registros de manera dinámica con funciones CONCAT y operadores CROSS JOIN. Finalmente, con su orientación, opté por la segunda alternativa, logrando así una carga completamente automatizada y escrita en SQL puro, sin depender de archivos externos.

Además, la IA me ayudó a optimizar la generación de datos, aprovechando expresiones condicionales CASE y funciones de fecha para distribuir valores realistas en los campos de estado, actividad y fechas de registro. También me sirvió para revisar la coherencia del modelo relacional, sugiriendo el uso de TRIGGERS y acciones en cascada (ON DELETE CASCADE) que luego adapté, probé y ajusté manualmente en MySQL Workbench.

Por último, la utilicé como apoyo para redactar las explicaciones técnicas incluidas en el informe, con el fin de describir con claridad el funcionamiento de cada bloque de código y justificar las decisiones de diseño adoptadas.

Todas las pruebas, verificaciones y ejecuciones fueron realizadas por mí en un entorno real de base de datos, analizando los resultados y aplicando los cambios necesarios. En resumen, la IA fue una herramienta de apoyo conceptual y de corrección, pero el desarrollo, la adaptación del código y la validación final del proyecto fueron completamente de mi autoría.

### **Prompt:**

Necesito que me expliques de manera breve y teórica, en tono académico, cómo es posible realizar la carga masiva de datos, optimización con CASE y funciones de fecha, coherencia

del modelo relacional con triggers y cascadas, y redacción técnica. Solo fundamentos teóricos

## 1. Verificación de sintaxis y estructura

En bases de datos relacionales, la sintaxis correcta asegura que las sentencias se ajusten al lenguaje SQL estándar. La estructura, en cambio, se refiere a que las claves primarias, foráneas y restricciones estén bien definidas para garantizar **integridad referencial** y **consistencia de datos**. Esto es lo que permite que el modelo sea confiable y no se generen registros inválidos.

### 1. Carga externa (BULK INSERT u operaciones equivalentes)

- **Fundamento:** se basa en la idea de que los datos ya existen en un archivo externo (CSV, TXT, etc.) y el motor de base de datos solo actúa como receptor.
- **Ventaja principal:** es extremadamente eficiente para grandes volúmenes porque el motor optimiza la lectura secuencial y evita la sobrecarga de múltiples sentencias individuales.
- **Implicación teórica:** aquí la base de datos funciona como un *sistema de ingestión masiva*, delegando la preparación de los datos a procesos externos. Se prioriza la **velocidad de transferencia** sobre la flexibilidad.
- **Limitación:** depende de la existencia de archivos externos y de que estos estén correctamente formateados. Además, introduce un punto de dependencia fuera del propio motor.

### 2. Generación interna (uso de operadores como CONCAT y CROSS JOIN)

- **Fundamento:** en lugar de importar datos, se generan dentro del propio motor mediante combinaciones y transformaciones.
- **Producto cartesiano:** el CROSS JOIN es la base teórica de este enfoque, porque permite multiplicar conjuntos de valores y, a partir de ellos, construir registros de manera dinámica.
- **Expresividad:** funciones como concatenación, condiciones lógicas o de fecha permiten simular variabilidad y realismo en los datos, sin necesidad de fuentes externas.
- **Implicación teórica:** aquí la base de datos no solo almacena, sino que también **produce datos sintéticos**. Se aprovecha su capacidad declarativa para crear escenarios de prueba controlados.
- **Ventaja:** independencia total de archivos externos, portabilidad del script y control absoluto sobre la lógica de generación.
- **Limitación:** puede ser menos eficiente que un BULK INSERT cuando el volumen es muy grande, porque el motor debe calcular todas las combinaciones en tiempo real.

### 3. Comparación conceptual

- **Carga externa** → privilegia la **eficiencia operativa** y la integración con sistemas externos.
- **Generación interna** → privilegia la **autonomía y reproducibilidad** dentro del propio motor.

#### 4. Optimización con expresiones y funciones

El uso de **CASE** y funciones de fecha responde a la idea de **simulación controlada de datos**. Teóricamente, esto permite distribuir valores de manera realista, evitando sesgos y asegurando que los datos de prueba reflejen escenarios variados.

#### 4. Coherencia del modelo relacional

Aquí entra la teoría de la **integridad referencial** y de las **acciones en cascada**. Los **TRIGGERS** son mecanismos que automatizan reglas de negocio, mientras que **ON DELETE CASCADE** asegura que las relaciones entre tablas se mantengan consistentes cuando se eliminan registros.

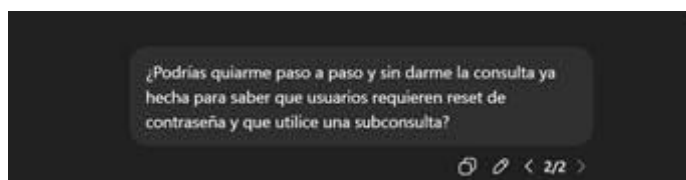
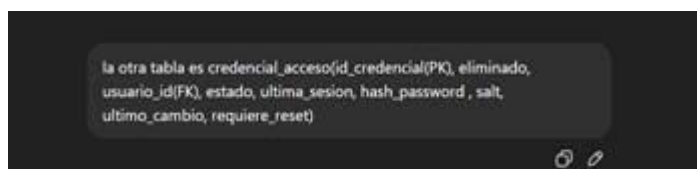
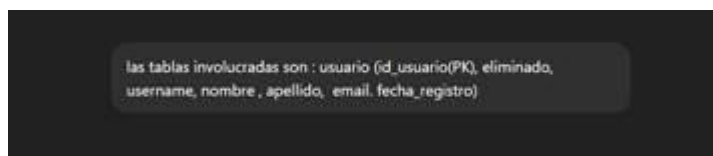
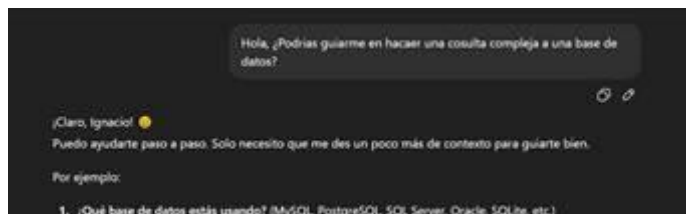


---

## Etapa 3. Consultas complejas y útiles a partir del CRUD inicial

Generé una consulta que contenga una subconsulta. La consulta se puede ver en el archivo de consultas.

Prompts:



Resumen de la respuesta obtenida:

### 1) Decide qué columnas vas a mostrar

Piensa qué datos del usuario quieres ver en el resultado. Ejemplos típicos:

- `id_usuario`
- `username`
- `email`
- `nombre, apellido`
- `fecha_registro`

Anótalos: esos serán los campos en la parte `SELECT`.

### 2) Define las reglas de filtro generales

Normalmente querrás excluir registros "eliminados". Entonces considera estas condiciones (en lenguaje natural):

### 3) Piensa el propósito de la subconsulta

La subconsulta debe devolver los usuarios que requieren reset. En términos lógicos:

- La subconsulta produce una lista (o booleano) que identifica usuarios con `requiere_reset = 1` (y que no estén marcados como eliminados).
- Luego la consulta principal usa esa lista para filtrar la tabla `usuario`.

Hay dos patrones comunes para usar la subconsulta:

- **IN:** `WHERE id_usuario IN (subconsulta_que_devuelve_usuario_id)` — sencillo y directo.
- **EXISTS** (subconsulta correlacionada): `WHERE EXISTS (subconsulta_que_comprueba_la_existencia_de_una_credencial_para_el_usuario_actual_con_requiere_reset = 1)` — útil si quieres chequear condición existencia por cada usuario y permitir lógica más compleja (por ejemplo, comprobar la última credencial).

Decide cuál prefieres (IN es más simple; EXISTS es más flexible y a veces más eficiente con índices).

### 4) Cómo escribir la subconsulta (en lógica, no en SQL completo)

La subconsulta debe:

- Buscar en `credencial_acceso`.
- Filtrar por `requiere_reset = 1`.
- Filtrar por `eliminado = 0` (si corresponde).
- Devolver `usuario_id` (la FK que liga a `usuario`).

En palabras:

```
"Selecciona los usuario_id de la tabla credencial_acceso donde requiere_reset = 1 y la credencial no esté eliminada."
```

Si usas **EXISTS**, la subconsulta será correlacionada: por cada fila de `usuario` comprobarás si existe al menos una fila en `credencial_acceso` que cumpla esas condiciones y tenga `usuario_id = usuario.id_usuario`.

## 9) Seguridad

- No incluya campos sensibles (p. ej. `hash_password`, `salt`) en el resultado ni en logs.
- Asegúrate de que el usuario que ejecuta la consulta tenga solo los permisos necesarios.

## 10) Ejemplo de pensamiento final (plantilla conceptual)

En tu cabeza arma algo como:

- `SELECT` [mis columnas de usuario]
- `FROM` usuario
- `WHERE` usuario.no\_eliminado  
AND (usuario.id\_usuario está en la lista devuelta por la subconsulta que busca en credencial\_acceso las filas con requiere\_reset=1 y no eliminadas)

O, alternativamente con `EXISTS`:

- `WHERE EXISTS` (comprobar en credencial\_acceso para este usuario que requiere\_reset=1 y credencial no eliminada)

---

## Etapa 4 – Seguridad e Integridad:

El primer punto de la metodología es la Creación de un usuario con permisos acotados y demostrar la regla de usuario con mínimos privilegios. Para cumplir con esto, necesitamos:

1. Crear una cuenta nueva (por ejemplo, usuario\_app).
2. Asignarle solo los permisos estrictamente necesarios (el mínimo privilegio) para que una aplicación pueda consultar los datos.

Pregunta Disparadora: ¿Qué metodos en SQL usarías para crear un usuario llamado usuario\_app y luego asignarle el permiso de lectura (SELECT) a una tabla de tu base de datos (por ejemplo, la tabla usuario)? Piensa en estos dos comandos clave: • Uno para crear la cuenta. • Otro para otorgar permisos específicos.

### Paso 1: Crear la cuenta

En SQL existe un comando específico para **dar de alta un nuevo usuario en el motor de base de datos**.

- La idea es: *“quiero un usuario nuevo con un nombre y una contraseña”*. CREATE USER

Con esto, el motor ya reconoce al usuario como alguien que puede intentar conectarse, pero todavía **no tiene permisos sobre ninguna base ni tabla**.

### Paso 2: Asignar permisos mínimos

El segundo comando clave es el que **otorga privilegios específicos**.

- Se llama GRANT.
- La lógica es: *“a este usuario le permito hacer tal acción sobre tal objeto”*.
- SELECT significa solo lectura.

Con esto, el usuario podrá **consultar datos** de esa tabla, pero no podrá modificarlos ni acceder a otras tablas.

## La idea de “mínimo privilegio”

- No se le da acceso global ni permisos de escritura.
- Solo lo estrictamente necesario: en este caso, leer una tabla concreta.
- Esto protege la base de datos frente a errores o ataques, porque aunque alguien robe esas credenciales, **no podría borrar ni alterar nada**.

## Que es un inyección sql y como se previene?

### ¿Qué es la inyección SQL?

La inyección SQL es una técnica de ataque en la que un usuario malintencionado aprovecha un campo de entrada (por ejemplo, un formulario de login o un parámetro en una consulta) para **insertar instrucciones SQL adicionales** que el sistema no debería ejecutar. El objetivo suele ser:

- Obtener información confidencial (usuarios, contraseñas, datos personales).
- Alterar datos (modificar, borrar, insertar registros).
- Incluso dañar la base de datos (borrar tablas completas).

### ¿Cómo se aprovecha?

El atacante busca **romper la lógica de la consulta original**. Ejemplo conceptual:

- El sistema espera un dato simple como un nombre o un hash.
- El atacante introduce un texto que incluye comillas, comandos SQL y comentarios.
- Si la aplicación construye la consulta concatenando texto, ese “dato” se convierte en parte del código SQL.

### ¿Por qué es peligroso?

Porque si la aplicación no valida ni protege la entrada:

- El atacante puede **leer datos que no debería ver** (ejemplo: todas las contraseñas).
- Puede **alterar la lógica de autenticación** (entrar sin credenciales válidas).
- Puede **ejecutar comandos destructivos** (borrar tablas, cambiar permisos).

### ¿Cómo se previene?

1. **Uso de parámetros** (también llamados *consultas preparadas* o *stored procedures*): El motor de la base de datos trata la entrada como **dato literal**, nunca como código.
2. **Validación de entradas**: Asegurarse de que los datos cumplen con el formato esperado (ejemplo: un email válido, un número en un rango).

3. **Principio de privilegios mínimos:** Los usuarios de base de datos que usa la aplicación no deberían tener permisos de administración (así, aunque hubiera un ataque, el daño es limitado).
4. **Monitoreo y auditoría:** Revisar logs y detectar intentos sospechosos.

**En resumen:** La inyección SQL es peligrosa porque convierte datos en código. La defensa está en separar siempre **qué es dato** y **qué es instrucción**, usando parámetros, validación y permisos mínimos.

---

## Etapa 5. Concurrencia y transacciones

### Deadlocks

#### Prompt:

“De que me sirve los índices para evitar deadlocks?”

#### Respuesta:

Los **índices** no eliminan mágicamente los deadlocks, pero sí ayudan a **reducir la probabilidad** de que ocurran. Te explico por qué:

#### Cómo influyen los índices en la concurrencia y los deadlocks

##### 1. Menos filas bloqueadas

- Sin índice, MySQL/InnoDB debe escanear muchas filas para encontrar las que cumplen la condición.
- Ese escaneo implica **bloquear más registros** (o incluso páginas completas), aumentando la chance de que otra transacción choque con esos locks.
- Con un índice bien diseñado, el motor localiza directamente las filas necesarias y bloquea solo esas.

##### 2. Orden consistente de acceso

- Los deadlocks ocurren cuando dos transacciones toman locks en distinto orden.
- Los índices ayudan a que el motor acceda a las filas en un **orden predecible** (por ejemplo, siguiendo el índice por usuario\_id), reduciendo la probabilidad de ciclos de espera.

##### 3. Menor tiempo de retención de locks

- Una búsqueda indexada es más rápida que un full scan.
- Al terminar antes, la transacción mantiene los locks menos tiempo, lo que disminuye la ventana en la que puede producirse un deadlock.

#### Importante

- Los índices **no garantizan que no haya deadlocks**: siempre pueden ocurrir en sistemas concurrentes.
- Lo que hacen es **mitigar** el riesgo al reducir la contención y hacer más eficiente el acceso.
- Por eso, además de índices, se recomienda:

- Diseñar transacciones cortas.
- Acceder a las tablas en un orden consistente.
- Manejar deadlocks con reintentos (como ya implementaste en tus procedimientos).

---

## Crear un procedimiento almacenado en SQL

### **Prompt:**

“Podrías ayudarme a crear un procedimiento almacenado , quiero cambiar si un usuario se encuentra activo, pero evitando deadlocks en caso de que otra sesion intente hacer lo mismo. Recordá no darme código solo asistirme a crear algo así”

### **Respuesta:**

#### **¿Qué es un procedimiento almacenado?**

Es un bloque de instrucciones SQL que queda guardado en la base de datos con un nombre. Podés llamarlo como si fuera una función, pasándole parámetros de entrada o salida. Sirve para encapsular lógica de negocio directamente en el motor.

#### **Cómo funciona este procedimiento en particular**

- 1. Recibe parámetros de entrada**
  - El `id_usuario` que querés modificar.
  - El nuevo valor de activo (`TRUE/FALSE`).
- 2. Declara variables internas**
  - Contadores de intentos.
  - Flags para saber si hubo éxito o si se detectó un deadlock.
  - Variables para guardar información de error (código, estado SQL, mensaje).
- 3. Define un handler de errores**
  - Un *handler* es como un “capturador” de excepciones.
  - En este caso, está preparado para atrapar el error de deadlock (`SQLSTATE 40001`).
  - Cuando ocurre, hace un rollback, registra el error en una tabla de logs y marca que hubo un deadlock.



#### 4. Implementa un bucle de reintentos

- Intenta ejecutar la transacción (el UPDATE sobre la tabla usuario).
- Si no hubo deadlock, confirma (COMMIT) y marca éxito.
- Si hubo deadlock, vuelve a intentar hasta un máximo de veces definido.

#### 5. Manejo final de errores

- Si después de todos los intentos no se pudo completar, registra un error definitivo en la tabla de logs.
- Luego lanza un SIGNAL (como una excepción) para avisar al cliente que la operación falló.
- Si el error no fue un deadlock, también hace rollback y lanza un mensaje genérico.

### En resumen

Este procedimiento es un **patrón de robustez**:

- Intenta actualizar un campo (activo) de forma transaccional.
- Si ocurre un deadlock, lo detecta, lo registra y reintenta.
- Si después de varios intentos no se logra, deja evidencia en logs y devuelve un error controlado.

## Indices en entornos concurrentes

### Prompt:

“Como mido impacto de índices en entornos concurrentes. en que me ayudaría un índice en estos casos? que diferencia harian?”

### Respuesta:

Qué significa medir el impacto de índices en concurrencia?

Cuando varias sesiones acceden a la misma tabla al mismo tiempo (lecturas, escrituras, actualizaciones), el motor de base de datos necesita **localizar filas** y **bloquearlas** para mantener la integridad.

- **Sin índice:** el motor recorre toda la tabla (full table scan) y puede terminar bloqueando más filas de las necesarias.
- **Con índice:** el motor va directo a las filas relevantes y bloquea solo esas.

Medir el impacto implica observar **qué cambia en el comportamiento del sistema** cuando existe un índice en la columna usada en las condiciones (WHERE).

¿Qué observarías al medir?

1. **Tiempo de respuesta**
  - Con índice: las consultas y actualizaciones se resuelven más rápido.
  - Sin índice: tardan más porque se revisa toda la tabla.
2. **Bloqueos (locks)**
  - Con índice: se bloquean únicamente las filas que cumplen la condición.
  - Sin índice: se pueden bloquear rangos más amplios o incluso toda la tabla, aumentando la contención.
3. **Esperas y deadlocks**
  - Sin índice: mayor probabilidad de que dos transacciones se estorben entre sí, porque cada una bloquea más de lo necesario.
  - Con índice: menor riesgo, porque cada transacción toca solo lo que necesita.
4. **Escalabilidad**
  - Con índice: el sistema soporta más usuarios concurrentes sin degradarse.
  - Sin índice: a medida que crece la concurrencia, los tiempos de espera y los conflictos aumentan.

¿En qué te ayuda un índice en concurrencia?

- **Precisión en el acceso:** localiza exactamente las filas que buscás.
- **Menos bloqueos innecesarios:** reduce la “zona de impacto” de cada transacción.
- **Mayor paralelismo:** varias sesiones pueden trabajar en paralelo sobre distintas filas sin molestarse.
- **Menos riesgo de deadlocks:** al reducir la cantidad de filas bloqueadas, disminuye la probabilidad de ciclos de espera.

Diferencia clave

- **Sin índice:** el motor “barre” la tabla, bloquea de más, genera esperas y conflictos.
- **Con índice:** el motor “salta” directo a la fila, bloquea lo justo, y permite que la concurrencia sea más fluida.

En resumen: medir el impacto de índices en entornos concurrentes es comparar **tiempos, bloqueos y conflictos** con y sin índice. La diferencia fundamental es que el índice convierte operaciones costosas y conflictivas en operaciones rápidas y localizadas, lo que mejora la **eficiencia y la convivencia entre transacciones**.

## Fuentes de investigación:

Reglas de negocio:

<https://www.ibm.com/docs/en/iis/11.7.0?topic=types-business-rules>

Largo de un hash para un password:

[https://cheatsheetseries.owasp.org/cheatsheets/Password\\_Storage\\_Cheat\\_Sheet.html](https://cheatsheetseries.owasp.org/cheatsheets/Password_Storage_Cheat_Sheet.html)

Read select y Repeteable read:

<https://www.cockroachlabs.com/blog/sql-isolation-levels-explained/>