

## TECNICATURA UNIVERSITARIA EN PROGRAMACIÓN A DISTANCIA

### Bases de Datos I

#### **Trabajo Final Integrador**

#### **Modelado e Implementación de Base de Datos para el Sistema de “Gestión de empleados”**

#### **Integrantes**

Giardini Silvia	Comisión: 07
Marina Giselle Cordero	Comisión: 07
Alex Dauria	Comisión: 06
Matías Perdigués	Comisión: 01

#### **Profesores**

Nicolás Carcaño	Comisión: 07
Verónica Tomich	Comisión: 06
Guillermo Londero	Comisión: 01

**Fecha de Entrega**  
23 de Octubre de 2025

## Índice

<b>Introducción</b>	3
<b>Etapa 1: Modelado y Definición de Constraints</b>	4
<b>Etapa 2: Generación de Datos Masivos con SQL Puro</b>	7
<b>Etapa 3: Consultas complejas y útiles a partir del CRUD inicial</b>	15
<b>Etapa 4: Seguridad e Integridad</b>	25
<b>Etapa 5: Concurrencia y transacciones</b>	32
<b>Conclusión Final</b>	46

**Link del video explicativo**

<https://youtu.be/JJ2F1CTKpUg>

**Link del repositorio**

[https://github.com/Alex-Dauria/TrabajoFinalIntegrador\\_BDI](https://github.com/Alex-Dauria/TrabajoFinalIntegrador_BDI)

## Introducción

El presente Trabajo Final Integrador detalla el diseño, implementación y validación de una base de datos relacional para el sistema de “Gestión de Empleados”. Partiendo del modelado conceptual y la definición de reglas de negocio, se construyó un esquema SQL robusto, garantizando la integridad de los datos mediante el uso de constraints. El proyecto abarcó la carga masiva de un volumen significativo de registros para simular un entorno realista. Posteriormente, se desarrollaron consultas complejas y vistas para la generación de reportes, se aplicaron políticas de seguridad y, finalmente, se analizaron escenarios de concurrencia para asegurar el correcto funcionamiento transaccional del sistema.

## Etapa 1 – Modelado y Definición de Constraints

A continuación, se presenta el desarrollo de la primera etapa del Trabajo Final Integrador, centrada en el modelado de la base de datos para nuestro sistema de “**Gestión de empleados**”.

Partiendo de las reglas de negocio que definen el funcionamiento del sistema, se elaboró el Diagrama Entidad-Relación (DER) para representar conceptualmente los datos. Posteriormente, este diagrama fue traducido al modelo relacional y materializado en un script SQL que incluye todas las restricciones de integridad (PRIMARY KEY, FOREIGN KEY, UNIQUE y CHECK) necesarias para garantizar la consistencia y validez de la información. [ver: “01.esquema.sql”]

### Reglas de Negocio

Basado en el modelo de datos, el sistema se rige por las siguientes reglas de negocio:

#### Reglas Estructurales y de Relación (1:1)

- Cada Empleado debe tener un único Legajo asociado.
- Cada Legajo debe estar asociado a un único Empleado.

#### Reglas de Integridad para la Entidad Empleados

- Cada empleado debe tener un nombre, apellido, DNI y email. Estos campos son obligatorios.
- El DNI de cada empleado debe ser único en todo el sistema para evitar duplicados.
- Todo empleado debe tener una fecha de ingreso y un área asignada.

#### Reglas de Integridad para la Entidad Legajos

- Todo legajo debe estar asociado a un empleado\_id válido y existente. No pueden existir legajos sin un empleado.
- El número de legajo (nro\_legajo) debe ser único para cada registro, ya que es el identificador formal del empleado en la empresa.
- Cada legajo debe tener una categoría, un estado y una fecha de alta.
- El campo “estado” solo puede aceptar valores de una lista predefinida (por ejemplo: ‘Activo’, ‘Inactivo’).

### Diagrama DER:



### Script de creación del esquema:

```

CREATE TABLE empleados (
    id BIGINT AUTO_INCREMENT PRIMARY KEY,
    eliminado BOOLEAN NOT NULL DEFAULT FALSE,
    nombre VARCHAR(80) NOT NULL,
    apellido VARCHAR(80) NOT NULL,
    dni VARCHAR(15) NOT NULL UNIQUE,
    email VARCHAR(120) CHECK (email LIKE '%@%.%'),
    fecha_ingreso DATE,
    area VARCHAR(50)
);

CREATE TABLE legajos (
    id BIGINT AUTO_INCREMENT PRIMARY KEY,
    eliminado BOOLEAN NOT NULL DEFAULT FALSE,
    nro_legajo VARCHAR(20) NOT NULL UNIQUE,
    categoria VARCHAR(30),
    estado ENUM('ACTIVO', 'INACTIVO') NOT NULL,
    fecha_alta DATE,
    observaciones VARCHAR(255),
    empleado_id BIGINT NOT NULL UNIQUE,
    CONSTRAINT fk_empleado FOREIGN KEY (empleado_id) REFERENCES empleados(id) ON DELETE CASCADE ON UPDATE CASCADE
);

```

## Validación Práctica de las Restricciones

### Inserciones Correctas:

```
-- Inserción válida 1:
INSERT INTO empleados (eliminado, nombre, apellido, dni, email, fecha_ingreso, area)
VALUES (FALSE, 'Juan', 'Perez', '12345678', 'juan.perez@empresa.com', '2023-01-15', 'Recursos Humanos');

INSERT INTO legajos (eliminado, nro_legajo, categoria, estado, fecha_alta, observaciones, empleado_id)
VALUES (FALSE, 'LEG-001', 'Senior', 'ACTIVO', '2023-01-01', 'Legajo activo', 1);

-- Inserción válida 2:
INSERT INTO empleados (eliminado, nombre, apellido, dni, email, fecha_ingreso, area)
VALUES (FALSE, 'Maria', 'Gomez', '87654321', 'maria.gomez@empresa.com', '2024-06-20', 'TI');

INSERT INTO legajos (eliminado, nro_legajo, categoria, estado, fecha_alta, observaciones, empleado_id)
VALUES (FALSE, 'LEG-002', 'Junior', 'ACTIVO', '2024-06-01', 'Legajo Incompleto', 2);
```

### Inserciones Erróneas:

```
-- Error 1: Viola UNIQUE (nro_legajo duplicado)
INSERT INTO legajos (eliminado, nro_legajo, categoria, estado, fecha_alta, observaciones, empleado_id)
VALUES (FALSE, 'LEG-001', 'Analista', 'ACTIVO', '2024-10-01', 'Duplicado', 1); -- ya existe

-- Error 2: Viola FK (id no existe en empleados)
INSERT INTO legajos (eliminado, nro_legajo, categoria, estado, fecha_alta, observaciones, empleado_id)
VALUES (FALSE, 'LEG-003', 'Consultor', 'ACTIVO', '2024-10-01', 'Falta Documentación', 80); -- 80 aún no existe en empleados
```



## Etapa 2 – Generación de Datos Masivos con SQL Puro

En esta etapa se amplió el proyecto con la generación de datos ficticios en gran volumen, utilizando exclusivamente SQL, sin recurrir a otros lenguajes de programación. El objetivo fue poblar las tablas de empleados y legajos con miles de registros, simulando un escenario más cercano al funcionamiento real de un sistema de gestión.

### Objetivos:

- Comprender cómo poblar bases de datos solo con SQL.
- Evaluar el impacto del volumen de datos en tiempos de respuesta.
- Experimentar con la creación y el uso de índices.

### Descripción conceptual del mecanismo elegido

En esta etapa se implementó un proceso de generación masiva de datos utilizando **exclusivamente instrucciones SQL**, con el objetivo de poblar las tablas **empleados** y **legajos** con un volumen realista de registros (aproximadamente **450.000 filas en total**).

El propósito fue disponer de un conjunto de datos consistente y de gran tamaño que permita, en etapas posteriores, analizar rendimiento, índices y concurrencia sobre un entorno similar al de una base de datos en producción.

### Explicación detallada del script de generación masiva

El siguiente proceso tiene como finalidad poblar la base de datos **empresa** con un conjunto de registros masivo y coherente, garantizando integridad, unicidad y correspondencia 1:1 entre empleados y legajos.

Para ello se aplicaron técnicas de generación secuencial, pseudoaleatoriedad controlada y validaciones posteriores, utilizando únicamente SQL estándar.

#### 1. Determinación de valores máximos previos

Antes de realizar la inserción masiva, se estableció una etapa de preparación que obtiene los valores máximos actuales en las tablas **empleados** y **legajos**.

Esto permite **evitar duplicados** y mantener la continuidad de los identificadores en caso de futuras cargas adicionales.

```
-- Primero obtenemos el máximo ID y LEGAJO para evitar duplicados
SET @max_empleado_id = (SELECT COALESCE(MAX(id), 0) FROM empleados);
SET @max_legajo_num = (SELECT COALESCE(MAX(CAST(SUBSTRING(nro_legajo, 5) AS UNSIGNED)), 0) FROM legajos);
```

## 2. Inserción masiva de empleados

En esta etapa se generan **450.000 empleados nuevos** mediante una instrucción **INSERT ... SELECT** que combina seis subconsultas numéricas unidas por **CROSS JOIN**.

Cada combinación de dígitos (0–9) genera un identificador único (**seq.num**), que actúa como base para derivar todos los datos del empleado.

```
-- Insertar 450.000 empleados
INSERT INTO empleados (nombre, apellido, dni, email, fecha_ingreso, area)
SELECT
    CONCAT('Nombre_', LPAD(seq.num, 6, '0')),
    CONCAT('Apellido_', LPAD(seq.num, 6, '0')),
    LPAD(40000000 + @max_empleado_id + seq.num, 8, '0'),
    CONCAT('empleado', @max_empleado_id + seq.num, '@empresa.com'),
    DATE_ADD('2015-01-01', INTERVAL FLOOR(RAND() * 3650) DAY),
    ELT(FLOOR(1 + (RAND() * 5)), 'Ventas', 'RRHH', 'IT', 'Logística', 'Marketing')
```

```

) FROM (
    SELECT
        (a.n * 100000 + b.n * 10000 + c.n * 1000 + d.n * 100 + e.n * 10 + f.n) AS num
    FROM
        (SELECT 0 n UNION ALL SELECT 1 UNION ALL SELECT 2 UNION ALL SELECT 3 UNION ALL
         SELECT 4 UNION ALL SELECT 5 UNION ALL SELECT 6 UNION ALL SELECT 7 UNION ALL
         SELECT 8 UNION ALL SELECT 9) a
    CROSS JOIN
        (SELECT 0 n UNION ALL SELECT 1 UNION ALL SELECT 2 UNION ALL SELECT 3 UNION ALL
         SELECT 4 UNION ALL SELECT 5 UNION ALL SELECT 6 UNION ALL SELECT 7 UNION ALL
         SELECT 8 UNION ALL SELECT 9) b
    CROSS JOIN
        (SELECT 0 n UNION ALL SELECT 1 UNION ALL SELECT 2 UNION ALL SELECT 3 UNION ALL
         SELECT 4 UNION ALL SELECT 5 UNION ALL SELECT 6 UNION ALL SELECT 7 UNION ALL
         SELECT 8 UNION ALL SELECT 9) c
    CROSS JOIN
        (SELECT 0 n UNION ALL SELECT 1 UNION ALL SELECT 2 UNION ALL SELECT 3 UNION ALL
         SELECT 4 UNION ALL SELECT 5 UNION ALL SELECT 6 UNION ALL SELECT 7 UNION ALL
         SELECT 8 UNION ALL SELECT 9) d
    CROSS JOIN
        (SELECT 0 n UNION ALL SELECT 1 UNION ALL SELECT 2 UNION ALL SELECT 3 UNION ALL
         SELECT 4 UNION ALL SELECT 5 UNION ALL SELECT 6 UNION ALL SELECT 7 UNION ALL
         SELECT 8 UNION ALL SELECT 9) e
    CROSS JOIN
        (SELECT 0 n UNION ALL SELECT 1 UNION ALL SELECT 2 UNION ALL SELECT 3 UNION ALL
         SELECT 4 UNION ALL SELECT 5 UNION ALL SELECT 6 UNION ALL SELECT 7 UNION ALL
         SELECT 8 UNION ALL SELECT 9) f
) seq
WHERE seq.num BETWEEN 1 AND 450000;
```

Este bloque crea una **tabla semilla virtual** con más de un millón de combinaciones posibles.

El filtro **WHERE seq.num BETWEEN 1 AND 450000** limita el conjunto para obtener exactamente 450.000 filas.

Las funciones **CONCAT** y **LPAD** generan nombres y DNI's únicos.

**DATE\_ADD()** y **RAND()** asignan fechas de ingreso aleatorias entre 2015 y 2025.

**ELT()** selecciona aleatoriamente una de las cinco áreas laborales, simulando una distribución natural del personal.

Result Grid   Filter Rows:   Edit:   Export/Import:   Wrap Cell Content:   Fetch rows:								
	id	eliminado	nombre	apellido	dni	email	fecha_ingreso	area
4	0		Nombre_262000	Apellido_262000	40262002	empleado262002@empresa.com	2023-04-07	RRHH
5	0		Nombre_162000	Apellido_162000	40162002	empleado162002@empresa.com	2020-10-08	RRHH
6	0		Nombre_062000	Apellido_062000	40062002	empleado62002@empresa.com	2020-04-05	Marketing
7	0		Nombre_372000	Apellido_372000	40372002	empleado372002@empresa.com	2023-01-26	IT
8	0		Nombre_272000	Apellido_272000	40272002	empleado272002@empresa.com	2021-01-13	Marketing
9	0		Nombre_172000	Apellido_172000	40172002	empleado172002@empresa.com	2017-01-25	Logística
10	0		Nombre_072000	Apellido_072000	40072002	empleado72002@empresa.com	2019-12-22	Logística
11	0		Nombre_382000	Apellido_382000	40382002	empleado382002@empresa.com	2020-11-26	Ventas
12	0		Nombre_282000	Apellido_282000	40282002	empleado282002@empresa.com	2022-06-12	IT
13	0		Nombre_182000	Apellido_182000	40182002	empleado182002@empresa.com	2023-05-31	Marketing
14	0		Nombre_082000	Apellido_082000	40082002	empleado82002@empresa.com	2017-11-22	IT
15	0		Nombre_392000	Apellido_392000	40392002	empleado392002@empresa.com	2024-03-17	Marketing
16	0		Nombre_292000	Apellido_292000	40292002	empleado292002@empresa.com	2024-01-06	Logística
17	0		Nombre_192000	Apellido_192000	40192002	empleado192002@empresa.com	2023-06-25	Ventas
18	0		Nombre_092000	Apellido_092000	40092002	empleado92002@empresa.com	2015-01-23	Logística
19	0		Nombre_401000	Apellido_401000	40401002	empleado401002@empresa.com	2019-10-14	RRHH
20	0		Nombre_301000	Apellido_301000	40301002	empleado301002@empresa.com	2015-05-28	RRHH
21	0		Nombre_201000	Apellido_201000	40201002	empleado201002@empresa.com	2019-08-04	RRHH
22	0		Nombre_101000	Apellido_101000	40101002	empleado101002@empresa.com	2018-08-16	Logística
23	0		Nombre_001000	Apellido_001000	40001002	empleado1002@empresa.com	2022-07-29	IT
24	0		Nombre_411000	Apellido_411000	40411002	empleado411002@empresa.com	2016-05-04	RRHH
25	0		Nombre_311000	Apellido_311000	40311002	empleado311002@empresa.com	2022-04-14	Marketing
26	0		Nombre_211000	Apellido_211000	40211002	empleado211002@empresa.com	2021-06-02	RRHH
27	0		Nombre_111000	Apellido_111000	40111002	empleado111002@empresa.com	2021-06-08	RRHH
28	0		Nombre_011000	Apellido_011000	40011002	empleado11002@empresa.com	2019-07-12	IT
...	...	...	...	...	...	...	...	...
<b>empleados 1</b> 								

### 3. Inserción masiva de legajos (relación 1:1)

Luego se crean los legajos correspondientes a cada empleado.

Para mantener la correspondencia exacta (1 empleado = 1 legajo), se toma el mismo **id** de la tabla **empleados** y se utiliza la variable **@legajo\_counter** para generar los números de legajo consecutivos.

```
-- Insertar legajos - ESPECIFICANDO EL ID EXPLÍCITAMENTE
• SET @legajo_counter = @max_legajo_num;

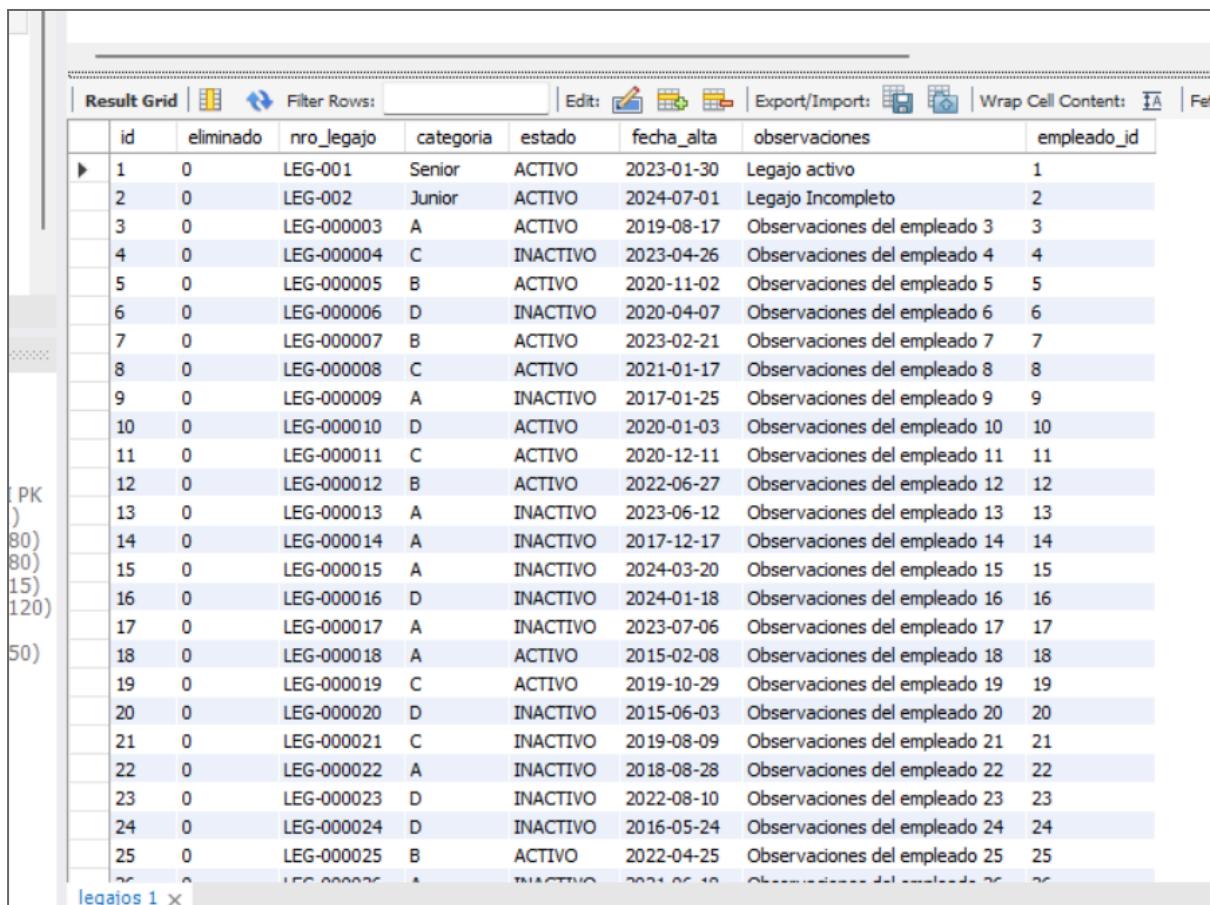
• INSERT INTO legajos (id, nro_legajo, categoria, estado, fecha_alta, observaciones, empleado_id)
SELECT
    e.id, -- Usamos el mismo ID del empleado
    CONCAT('LEG-', LPAD(@legajo_counter := @legajo_counter + 1, 6, '0')),
    ELT(FLOOR(1 + (RAND() * 4)), 'A', 'B', 'C', 'D'),
    ELT(FLOOR(1 + (RAND() * 2)), 'ACTIVO', 'INACTIVO'),
    DATE_ADD(e.fecha_ingreso, INTERVAL FLOOR(RAND() * 30) DAY), -- Usamos de referencia la de ingreso para que sea posterior
    CONCAT('Observaciones del empleado ', e.id),
    e.id
FROM empleados e
WHERE e.id > @max_empleado_id
ORDER BY e.id;
```

**@legajo\_counter** asegura que la numeración continúe correctamente.

Se usa **e.id** como clave principal y **empleado\_id** como clave foránea, preservando la relación 1:1.

La **fecha\_alta** del legajo se genera como la fecha de ingreso del empleado más un número aleatorio de días (entre 1 y 30), garantizando coherencia temporal.

Las categorías y estados se distribuyen aleatoriamente para representar diversidad en la base.



The screenshot shows a database grid titled 'Result Grid' with the following columns: id, eliminado, nro\_legajo, categoria, estado, fecha\_alta, observaciones, and empleado\_id. The data is as follows:

	id	eliminado	nro_legajo	categoria	estado	fecha_alta	observaciones	empleado_id
▶	1	0	LEG-001	Senior	ACTIVO	2023-01-30	Legajo activo	1
	2	0	LEG-002	Junior	ACTIVO	2024-07-01	Legajo Incompleto	2
	3	0	LEG-000003	A	ACTIVO	2019-08-17	Observaciones del empleado 3	3
	4	0	LEG-000004	C	INACTIVO	2023-04-26	Observaciones del empleado 4	4
	5	0	LEG-000005	B	ACTIVO	2020-11-02	Observaciones del empleado 5	5
	6	0	LEG-000006	D	INACTIVO	2020-04-07	Observaciones del empleado 6	6
	7	0	LEG-000007	B	ACTIVO	2023-02-21	Observaciones del empleado 7	7
	8	0	LEG-000008	C	ACTIVO	2021-01-17	Observaciones del empleado 8	8
	9	0	LEG-000009	A	INACTIVO	2017-01-25	Observaciones del empleado 9	9
	10	0	LEG-000010	D	ACTIVO	2020-01-03	Observaciones del empleado 10	10
	11	0	LEG-000011	C	ACTIVO	2020-12-11	Observaciones del empleado 11	11
	12	0	LEG-000012	B	ACTIVO	2022-06-27	Observaciones del empleado 12	12
	13	0	LEG-000013	A	INACTIVO	2023-06-12	Observaciones del empleado 13	13
	14	0	LEG-000014	A	INACTIVO	2017-12-17	Observaciones del empleado 14	14
	15	0	LEG-000015	A	INACTIVO	2024-03-20	Observaciones del empleado 15	15
	16	0	LEG-000016	D	INACTIVO	2024-01-18	Observaciones del empleado 16	16
	17	0	LEG-000017	A	INACTIVO	2023-07-06	Observaciones del empleado 17	17
	18	0	LEG-000018	A	ACTIVO	2015-02-08	Observaciones del empleado 18	18
	19	0	LEG-000019	C	ACTIVO	2019-10-29	Observaciones del empleado 19	19
	20	0	LEG-000020	D	INACTIVO	2015-06-03	Observaciones del empleado 20	20
	21	0	LEG-000021	C	INACTIVO	2019-08-09	Observaciones del empleado 21	21
	22	0	LEG-000022	A	INACTIVO	2018-08-28	Observaciones del empleado 22	22
	23	0	LEG-000023	D	INACTIVO	2022-08-10	Observaciones del empleado 23	23
	24	0	LEG-000024	D	INACTIVO	2016-05-24	Observaciones del empleado 24	24
	25	0	LEG-000025	B	ACTIVO	2022-04-25	Observaciones del empleado 25	25

#### 4. Verificaciones y controles de consistencia

Finalizada la carga, se ejecutan varias consultas para validar que el proceso haya sido correcto y que las restricciones se cumplan.

##### Conteo general de registros y verificación de integridad referencial

```

64 •   SELECT
65     (SELECT COUNT(*) FROM empleados) AS total_empleados,
66     (SELECT COUNT(*) FROM legajos) AS total_legajos,
67     (SELECT COUNT(*) FROM legajos WHERE empleado_id NOT IN (SELECT id FROM empleados)) AS FK_huerfanas;

```

	total_empleados	total_legajos	FK_huerfanas
▶	450002	450002	0

El siguiente bloque de consulta permite realizar una **verificación general de consistencia e integridad referencial** luego de la generación masiva de registros.

A través de tres subconsultas, se validan los siguientes aspectos:

1. **total\_empleados**: contabiliza el número total de registros generados en la tabla *empleados*.
2. **total\_legajos**: confirma que la cantidad de legajos coincide con la de empleados, asegurando la relación 1:1 entre ambas tablas.
3. **FK\_huerfanas**: detecta posibles registros inconsistentes en la tabla *legajos*, es decir, aquellos cuyo *empleado\_id* no existe en la tabla *empleados*.

El resultado obtenido muestra **450 002 empleados** y **450 002 legajos**, con **0 claves foráneas huérfanas**, lo que confirma que la carga masiva se ejecutó correctamente y que las **restricciones de integridad referencial** definidas en el modelo se mantienen vigentes y sin errores.

## Distribución de empleados por área

```

71 •   SELECT
72     area,
73     COUNT(*) AS cantidad,
74     ROUND(COUNT(*) * 100.0 / (SELECT COUNT(*) FROM empleados), 2) AS porcentaje
75   FROM empleados
76   GROUP BY area
77   ORDER BY porcentaje DESC;
78
79 -- Fechas coherentes (debe dar 0)

```

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

	area	cantidad	porcentaje
▶	Logística	90378	20.08
	RRHH	90179	20.04
	Marketing	89920	19.98
	Ventas	89910	19.98
	IT	89615	19.91

Esta consulta realiza un conteo agrupado de los empleados según el área a la que pertenecen, mostrando tanto la **cantidad absoluta** como el **porcentaje proporcional** que representa cada área sobre el total general.

Se utiliza la función `COUNT()` para contabilizar los registros y `ROUND()` para calcular el porcentaje con dos decimales, ordenando los resultados de mayor a menor participación.

Los datos obtenidos indican una **distribución equilibrada** entre las diferentes áreas de la organización, con valores cercanos al 20 % en cada una. Esto confirma que la generación masiva de datos se realizó con una **distribución uniforme**, tal como se planificó en la etapa de diseño.

Este tipo de consulta es útil para verificar proporciones y detectar posibles sesgos en la carga de información o en la generación automática de registros.

## Control de fechas entre empleado y legajo

```

79      -- Fechas coherentes (debe dar 0)
80 •  SELECT COUNT(*) AS legajos_con_fecha_alta_menor
81  FROM legajos l
82  JOIN empleados e ON l.empleado_id = e.id
83  WHERE l.fecha_alta < e.fecha_ingreso;

```

Result Grid	
	Filter Rows: <input type="text"/>
	Export:
	Wrap Cell Content:
legajos_con_fecha_alta_menor	0

Esta consulta verifica la **consistencia temporal** entre las fechas de alta del legajo y las fechas de ingreso del empleado.

El objetivo es garantizar que ningún legajo tenga una *fecha de alta anterior a la fecha de ingreso* de su correspondiente empleado, ya que esto representaría un error lógico en la relación de datos.

Para ello, se realiza una unión ([JOIN](#)) entre las tablas *empleados* y *legajos*, comparando ambos campos de fecha. Si la condición `l.fecha_alta < e.fecha_ingreso` se cumple, la consulta contaría esos registros inconsistentes.

El resultado obtenido es **0 registros con fecha incoherente**, lo que confirma que todas las relaciones entre empleados y legajos respetan la **integridad temporal** definida en el modelo de datos.

## Síntesis del proceso

Este procedimiento automatiza completamente la generación de un conjunto masivo de datos coherentes y reproducibles.

Las variables iniciales permiten mantener continuidad en futuras ejecuciones, las funciones de combinación (**CROSS JOIN**) y pseudoaleatoriedad (**RAND**, **ELT**) aseguran variedad y realismo, mientras que las verificaciones finales garantizan que se cumplan todas las **condiciones de integridad, cardinalidad y consistencia** del dominio.

**Resultado esperado:**

Un conjunto de 450.000 empleados y 450.000 legajos, enlazados 1:1, con datos válidos, únicos y distribuidos en diferentes áreas, categorías y estados.

**Script asociado:**

El script SQL que acompaña esta etapa se encuentra en el archivo [`03\_datos\_masivos.sql`](#), utilizado para la generación de 450 000 registros coherentes entre las tablas empleados y legajos.

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

### 1. Identificación de áreas que agreguen valor

Se identificaron cuatro áreas que agregarían valor al sistema

- No existe una **vista clara de la distribución de empleados por área**, lo que complica entender la estructura real del personal.
- No se pueden **analizar tendencias de contratación a lo largo del tiempo**, lo que limita la capacidad de planificar futuras incorporaciones.
- No hay ningún mecanismo que permita **detectar automáticamente inconsistencias**, por ejemplo, empleados sin legajo activo.
- Es complejo **segmentar empleados por experiencia o antigüedad**, lo cual es clave para diseñar planes de carrera y capacitación.

### 2. Diseño de consultas con técnicas avanzadas [ 05\_consultas.sql ]

#### CONSULTA 1

##### JOIN + GROUP BY + HAVING

Empleados por área con estadísticas

```

SELECT
    e.area,
    COUNT(*) AS total_empleados,
    COUNT(l.id) AS legajos_activos,
    AVG(DATEDIFF(CURDATE(), e.fecha_ingreso)) AS antiguedad_promedio_dias,
    COUNT(CASE WHEN l.estado = 'INACTIVO' THEN 1 END) AS legajos_inactivos
FROM empleados e
LEFT JOIN legajos l ON e.id = l.empleado_id
WHERE e.eliminado = FALSE
GROUP BY e.area
HAVING total_empleados > 1000 -- Enfocado en áreas significativas
ORDER BY total_empleados DESC;

```

**UTILIDAD:** Obtener un panorama general de la dotación de personal por departamento, identificando áreas con mayor concentración y analizando patrones de antigüedad y estado de legajos.

## CONSULTA 2

### INNER JOIN + GROUP BY múltiple + funciones de fecha

Tendencias de contratación por año y categoría

```

SELECT
    YEAR(l.fecha_alta) AS año,
    l.categoría,
    COUNT(*) AS total_legajos,
    COUNT(CASE WHEN l.estado = 'ACTIVO' THEN 1 END) AS activos,
    COUNT(CASE WHEN l.estado = 'INACTIVO' THEN 1 END) AS inactivos
FROM legajos l
INNER JOIN empleados e ON l.empleado_id = e.id
WHERE l.fecha_alta BETWEEN '2020-01-01' AND '2024-12-31'
    AND e.area IN ('IT', 'RRHH', 'Ventas') -- Áreas de alto impacto
    AND l.eliminado = FALSE
GROUP BY YEAR(l.fecha_alta), l.categoría
HAVING total_legajos > 50 -- Filtro de relevancia estadística
ORDER BY año DESC, total_legajos DESC;

```

**UTILIDAD:** Analizar cómo ha variado la contratación a lo largo del tiempo en áreas estratégicas, permitiendo identificar patrones estacionales y necesidades de planificación

## CONSULTA 3

### Subconsulta correlacionada + NOT EXISTS

Empleados sin legajo activo

```

SELECT
    e.id,
    e.nombre,
    e.apellido,
    e.area,
    e.fecha_ingreso
FROM empleados e
WHERE e.eliminado = FALSE
    AND NOT EXISTS (
        SELECT 1
        FROM legajos l
        WHERE l.empleado_id = e.id
            AND l.estado = 'ACTIVO'
            AND l.eliminado = FALSE
    )
ORDER BY e.fecha_ingreso DESC
LIMIT 100; -- Muestra los casos más recientes primero

```

**UTILIDAD:** Identifica empleados que no tienen legajo activo vigente, lo cual puede indicar problemas administrativos, personal en proceso de baja o errores de carga

#### CONSULTA 4

##### JOIN + CASE + GROUP BY múltiple + HAVING

Clasificación de empleados por antigüedad

```

SELECT
    e.area,
    CASE
        WHEN DATEDIFF(CURDATE(), e.fecha_ingreso) < 365 THEN 'Menos de 1 año'
        WHEN DATEDIFF(CURDATE(), e.fecha_ingreso) BETWEEN 365 AND 1825 THEN '1-5 años'
        WHEN DATEDIFF(CURDATE(), e.fecha_ingreso) > 1825 THEN 'Más de 5 años'
    END as rango_antiguedad,
    l.categoría,
    COUNT(*) as cantidad,
    ROUND(AVG(DATEDIFF(CURDATE(), e.fecha_ingreso)), 2) as antiguedad_promedio_dias
FROM empleados e
INNER JOIN legajos l ON e.id = l.empleado_id
WHERE e.eliminado = FALSE
    AND l.eliminado = FALSE
    AND l.estado = 'ACTIVO' -- Solo empleados actualmente activos
GROUP BY e.area, rango_antiguedad, l.categoría
HAVING cantidad > 10 -- Grupos con representatividad suficiente
ORDER BY e.area, cantidad DESC;

```

**UTILIDAD:** Segmentar la plantilla en rangos de experiencia para facilitar análisis de retención y detección de áreas con rotación alta o baja.

### 3. Prueba con grandes volúmenes de datos [ 05\_consultas.sql ]

#### CONSULTA 1

	area	total_empleados	legajos_activos	antiguedad_promedio_dias	legajos_inactivos
▶	Ventas	90151	90151	2109.9709	45195
	IT	90073	90073	2107.7012	45198
	RRHH	90050	90050	2111.6703	45155
	Marketing	89918	89918	2103.0115	45235
	Logística	89810	89810	2111.5644	44980

**Resultado:** 5 áreas principales con aproximadamente 90.000 empleados cada una, antigüedad promedio de 5.8 años.

#### CONSULTA 2

	año	categoria	total_legajos	activos	inactivos
▶	2024	A	6894	3438	3456
	2024	D	6840	3362	3478
	2024	C	6742	3338	3404
	2024	B	6601	3268	3333
	2023	C	6839	3416	3423
	2023	D	6798	3335	3463
	2023	A	6727	3335	3392
	2023	B	6653	3297	3356
	2022	D	6945	3425	3520
	2022	C	6860	3407	3453
	2022	B	6779	3354	3425
	2022	A	6641	3320	3321
	2021	B	6831	3305	3526
	2021	C	6782	3399	3383
	2021	A	6708	3371	3337
	2021	D	6559	3210	3349
	2020	D	6845	3470	3375
	2020	B	6826	3478	3348
	2020	C	6734	3375	3359
	2020	A	6645	3306	3339

**Resultado:** Distribución equilibrada entre categorías A, B, C, D con volúmenes similares cada año, mostrando aproximadamente 50% de empleados activos e inactivos en cada categoría.

### CONSULTA 3

	<b>id</b>	<b>nombre</b>	<b>apellido</b>	<b>area</b>	<b>fecha_ingreso</b>
	449303	Nombre_160899	Apellido_160899	RRHH	2024-12-28
	415287	Nombre_213974	Apellido_213974	RRHH	2024-12-28
	409478	Nombre_277970	Apellido_277970	Ventas	2024-12-28
	405293	Nombre_148857	Apellido_148857	RRHH	2024-12-28
	395360	Nombre_089850	Apellido_089850	Logística	2024-12-28
	392625	Nombre_227948	Apellido_227948	RRHH	2024-12-28
	389181	Nombre_072835	Apellido_072835	RRHH	2024-12-28
	384717	Nombre_267832	Apellido_267832	RRHH	2024-12-28
	370784	Nombre_283922	Apellido_283922	Ventas	2024-12-28
	353564	Nombre_251700	Apellido_251700	Ventas	2024-12-28
	343142	Nombre_182903	Apellido_182903	Ventas	2024-12-28
	341522	Nombre_256691	Apellido_256691	Logística	2024-12-28

**Resultado:** 100 empleados cuyos legajos están marcados como INACTIVOS, lo que es útil para el seguimiento de desvinculaciones o fallas administrativas.

*[La tabla muestra una representación parcial]*

### CONSULTA 4

	<b>area</b>	<b>rango_antiguedad</b>	<b>categoria</b>	<b>cantidad</b>	<b>antiguedad_promedio_dias</b>
	Logística	Menos de 1 año	D	223	326.58
	Marketing	Más de 5 años	D	6505	2873.15
	Marketing	Más de 5 años	B	6419	2866.20
	Marketing	Más de 5 años	C	6388	2882.08
	Marketing	Más de 5 años	A	6386	2870.59
	Marketing	1-5 años	A	4540	1093.80
	Marketing	1-5 años	D	4497	1095.01
	Marketing	1-5 años	C	4463	1089.30
	Marketing	1-5 años	B	4442	1093.66
	Marketing	Menos de 1 año	B	271	323.20
	Marketing	Menos de 1 año	A	260	323.56
	Marketing	Menos de 1 año	D	257	321.50
	Marketing	Menos de 1 año	C	255	325.04
	RRHH	Más de 5 años	D	6670	2872.93
	RRHH	Más de 5 años	B	6597	2888.39
	RRHH	Más de 5 años	A	6426	2884.23
	RRHH	Más de 5 años	C	6363	2873.96
	RRHH	1-5 años	C	4525	1095.07
	RRHH	1-5 años	D	4487	1097.42
	RRHH	1-5 años	A	4442	1097.64
	RRHH	1-5 años	B	4427	1100.88
	RRHH	Menos de 1 año	D	260	325.09

**Resultado:** Mayoría de empleados con +5 años de experiencia, distribución balanceada por categoría. *[La tabla muestra una representación parcial]*

#### 4. Medición de tiempos con y sin índices [ 04\_indices.sql ]

Para optimizar las consultas se crearon índices específicos:

##### Índices para consultas de IGUALDAD

```
CREATE INDEX idx_empleados_area ON empleados(area);
CREATE INDEX idx_legajos_estado ON legajos(estado);
CREATE INDEX idx_legajos_categoria ON legajos(categoría);
```

##### Índices para consultas de RANGO:

```
CREATE INDEX idx_empleados_fecha_ingreso ON empleados(fecha_ingreso);
CREATE INDEX idx_legajos_fecha_alta ON legajos(fecha_alta);
```

##### Índices compuestos para JOINS:

```
CREATE INDEX idx_empleados_area_fecha ON empleados(area, fecha_ingreso);
CREATE INDEX idx_legajos_estado_fecha ON legajos(estado, fecha_alta);
```

Se verificó que todos los índices se hayan creado exitosamente:

	TABLE_NAME	INDEX_NAME	COLUMNAS	INDEX_TYPE
▶	empleados	idx_empleados_area	area	BTREE
	empleados	idx_empleados_area_fecha	area,fecha_ingreso	BTREE
	empleados	idx_empleados_fecha_ingreso	fecha_ingreso	BTREE
	legajos	idx_legajos_categoria	categoría	BTREE
	legajos	idx_legajos_estado	estado	BTREE
	legajos	idx_legajos_estado_fecha	estado,fecha_alta	BTREE
	legajos	idx_legajos_fecha_alta	fecha_alta	BTREE

Se utilizaron tres tipos de consultas, cada una ejecutada cinco veces, analizando su desempeño con **EXPLAIN ANALYZE**.

Para cada consulta se evaluaron versiones **con** índice y **sin** índice (usando **IGNORE INDEX**), tomando la mediana de los tiempos para lograr un resultado más representativo.

## Ejemplos de resultados de ejecución [ 05\_explain.sql ]

```

EXPLAIN: -> Aggregate: count(0) (cost=5773 rows=1) (actual time=227..227 rows=1 loops=1)
          -> Filter: (empleados.eliminado = false) (cost=3980 rows=17925) (actual time=0.186..222 rows=90073 loops=1)
              -> Index lookup on empleados using idx_empleados_area (area='IT') (cost=3980 rows=179250) (actual time=0.185..215 rows=90073 loops=1)

EXPLAIN: -> Aggregate: count(0) (cost=46561 rows=1) (actual time=271..271 rows=1 loops=1)
          -> Filter: ((empleados.eliminado = false) and (empleados.area = 'IT')) (cost=45443 rows=11178) (actual time=0.0622..267 rows=90073 loops=1)
              -> Table scan on empleados (cost=45443 rows=447139) (actual time=0.0595..193 rows=450002 loops=1)

EXPLAIN: -> Aggregate: count(0) (cost=26547 rows=1) (actual time=247..247 rows=1 loops=1)
          -> Filter: ((legajos.estado = 'ACTIVO') and (legajos.fecha_alta between '2023-01-01' and '2024-12-31')) (cost=17745 rows=88020) (actual time=0.511..242 rows=44801 loops=1)
              -> Covering index range scan on legajos using idx_legajos_estado_fecha over (estado = 'ACTIVO' AND '2023-01-01' <= fecha_alta <= '2024-12-31') (cost=17745 rows=88020) (actual time=0.497..148 rows=44801 loops=1)

EXPLAIN: -> Aggregate: count(0) (cost=50392 rows=1) (actual time=525..525 rows=1 loops=1)
          -> Filter: ((legajos.estado = 'ACTIVO') and (legajos.fecha_alta between '2023-01-01' and '2024-12-31')) (cost=45415 rows=49770) (actual time=0.153..521 rows=44801 loops=1)
              -> Table scan on legajos (cost=45415 rows=447978) (actual time=0.146..242 rows=450002 loops=1)

EXPLAIN: -> Table scan on <temporary> (actual time=1673..1673 rows=5 loops=1)
          -> Aggregate using temporary table (actual time=1673..1673 rows=5 loops=1)
              -> Nested loop inner join (cost=102647 rows=111994) (actual time=0.617..1565 rows=111966 loops=1)
                  -> Index lookup on l using idx_legajos_estado (estado='ACTIVO'), with index condition: (l.estado = 'ACTIVO') (cost=24251 rows=223989) (actual time=0.603..853 rows=224239)

EXPLAIN: -> Table scan on <temporary> (actual time=9004..9004 rows=5 loops=1)
          -> Aggregate using temporary table (actual time=9004..9004 rows=5 loops=1)
              -> Nested loop inner join (cost=97604 rows=149031) (actual time=0.6..8712 rows=111966 loops=1)
                  -> Filter: (e.fecha_ingreso > DATE'2020-01-01') (cost=45443 rows=149031) (actual time=0.142..831 rows=224983 loops=1)

```

## Resultados y mediana tras cinco ejecuciones

### Consulta 1 - Igualdad

**Con índice:** 227 ms, 618 ms, 806 ms, 1519 ms, 1531 ms

**Mediana:** 806 ms

**Sin índice:** 271 ms, 614 ms, 620 ms, 658 ms, 743 ms

**Mediana:** 620 ms

### Consulta 2 - Rango

**Con índice:** 79.2 ms, 225 ms, 230 ms, 243 ms, 247 ms

**Mediana:** 230 ms

**Sin índice:** 525 ms, 903 ms, 904 ms, 935 ms, 952 ms

**Mediana:** 904 ms

### Consulta 3 - JOIN

**Con índice:** 1673 ms, 3643 ms, 3776 ms, 3862 ms, 3867 ms

**Mediana:** 3776 ms

**Sin índice:** 3884 ms, 8066 ms, 8844 ms, 8922 ms, 9004 ms

**Mediana:** 8844 ms

**Tabla Comparativa Final**

CONSULTA	CON ÍNDICE	SIN ÍNDICE	MEJORA
Igualdad	806 ms	620 ms	0.77x
Rango	230 ms	904 ms	3.93x
JOIN	3776 ms	8844 ms	2.34x

### Conclusión

Los índices demuestran un **impacto positivo** significativo en consultas de rango (3.93x más rápido) y JOINs complejos (2.34x más rápido), donde facilitan accesos ordenados y reducen escaneos completos.

Sin embargo, en consultas de igualdad simple, el índice **puede degradar** el rendimiento (0.77x) cuando no cubre todos los filtros aplicados, forzando costosos accesos adicionales a la tabla principal.

La efectividad de un índice depende críticamente de su selectividad y capacidad para cubrir completamente los criterios de consulta, destacando la importancia de analizar patrones de acceso específicos antes de su implementación.

### Implementación de Vistas para Reportes Recurrentes [ 06\_vistas.sql ]

Con el objetivo de simplificar el acceso a la información más relevante, se desarrollaron dos vistas que consolidan datos estratégicos del sistema.

Estas vistas permiten obtener reportes ejecutivos y análisis temporales de manera inmediata, evitando la necesidad de ejecutar consultas complejas frecuentemente.

### VISTA 1

Resumen ejecutivo de empleados activos por área

**Proporciona un dashboard instantáneo de métricas clave por área, ideal para reuniones de gerencia y planificación de recursos humanos.**



```
CREATE OR REPLACE VIEW vista_resumen_empleados_activos AS
SELECT
    e.area,
    COUNT(*) as total_empleados,
    COUNT(l.id) as legajos_activos,
    ROUND(AVG(DATEDIFF(CURDATE(), e.fecha_ingreso)/365), 2) as antiguedad_promedio_años,
    MIN(e.fecha_ingreso) as fecha_ingreso_mas_antigua,
    MAX(e.fecha_ingreso) as fecha_ingreso_mas_reciente,
    GROUP_CONCAT(DISTINCT l.categoría ORDER BY l.categoría SEPARATOR ', ') as categorías_presentes
FROM empleados e
LEFT JOIN legajos l ON e.id = l.empleado_id AND l.estado = 'ACTIVO' AND l.eliminado = FALSE
WHERE e.eliminado = FALSE
GROUP BY e.area;
```

## VISTA 2

Análisis temporal de altas y bajas

**Facilita el seguimiento de tendencias de contratación y desvinculación, permitiendo detectar patrones estacionales y evaluar la efectividad de políticas de reclutamiento.**

```
CREATE OR REPLACE VIEW vista_analisis_temporal AS
SELECT
    YEAR(l.fecha_alta) as año,
    MONTH(l.fecha_alta) as mes,
    l.categoría,
    e.area,
    COUNT(*) as total_movimientos,
    COUNT(CASE WHEN l.estado = 'ACTIVO' THEN 1 END) as altas,
    COUNT(CASE WHEN l.estado = 'INACTIVO' THEN 1 END) as bajas,
    ROUND(COUNT(CASE WHEN l.estado = 'ACTIVO' THEN 1 END) * 100.0 / COUNT(*), 2) as porcentaje_activos
FROM legajos l
INNER JOIN empleados e ON l.empleado_id = e.id
WHERE l.fecha_alta >= '2020-01-01'
    AND l.eliminado = FALSE
    AND e.eliminado = FALSE
GROUP BY YEAR(l.fecha_alta), MONTH(l.fecha_alta), l.categoría, e.area;
```



## EJEMPLOS DE APLICACIÓN

```
SELECT * FROM vista_resumen_empleados_activos WHERE total_empleados > 1000;  
SELECT * FROM vista_analisis_temporal WHERE año = 2024 AND area = 'IT';
```

## EJEMPLOS DE SALIDA

area	total_empleados	legajos_activos	antiguedad_promedio_años	fecha_ingreso_mas_antigua	fecha_ingreso_mas_reciente	categorias_presentes
Ventas	90151	44956	5.78	2015-01-01	2024-12-28	A, B, C, D
IT	90073	44875	5.78	2015-01-01	2024-12-28	A, B, C, D, Junior
RRHH	90050	44895	5.79	2015-01-01	2024-12-28	A, B, C, D, Senior
Marketing	89918	44683	5.76	2015-01-01	2024-12-28	A, B, C, D
Logística	89810	44830	5.79	2015-01-01	2024-12-28	A, B, C, D

	año	mes	categoria	area	total_movimientos	altas	bajas	porcentaje_activos
	2024	12	D	IT	197	106	91	53.81
	2024	12	B	IT	193	94	99	48.70
	2024	12	A	IT	193	107	86	55.44
	2024	12	C	IT	197	102	95	51.78
	2024	11	B	IT	173	77	96	44.51
	2024	11	C	IT	181	86	95	47.51
	2024	11	A	IT	190	91	99	47.89
	2024	11	D	IT	198	92	106	46.46
	2024	10	B	IT	182	82	100	45.05

Esta estructura conecta la parte de optimización (**índices**) con la de usabilidad (**vistas**), mostrando cómo todo se integra para crear un sistema más eficiente y usable.

## Etapa 4 - Seguridad e integridad [07\_seguridad.sql]

En este apartado se proponen una serie de medidas de seguridad tendientes a garantizar la integridad y la consistencia de nuestra base de datos haciendo, además, un breve recorrido sobre la base teórica que las sustentan. En este sentido, se emplearán medidas de autenticación segura, pondremos a prueba la consistencia de nuestra base de datos, especialmente evaluando la integridad referencial y de entidad y, por último, abordaremos un ejemplo de consulta parametrizada a fin de resguardar nuestra base de datos de intentos de SQL Injection.

### 1. Creación de un usuario con privilegios mínimos

Se creó el usuario '`gerente_marketing`'@'localhost' con contraseña segura:

```
USE empresa;
CREATE USER 'gerente_marketing'@'localhost' IDENTIFIED BY 'marketing131025';
```

Se otorgaron sólo permisos de **lectura y actualización** sobre **vistas limitadas**, no sobre las tablas reales:

```
GRANT SELECT ON empresa.empleados_marketing TO 'gerente_marketing'@'localhost';
GRANT UPDATE ON empresa.empleados_marketing TO 'gerente_marketing'@'localhost';
GRANT SELECT ON empresa.legajos_activos_marketing TO 'gerente_marketing'@'localhost';
```

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

	Grants for gerente_marketing@localhost
▶	GRANT USAGE ON *.* TO `gerente_marketing`@`localhost`
	GRANT SELECT, UPDATE ON `empresa`.`empleados_marketing` TO `gerente_marketing`@`localhost`
	GRANT SELECT ON `empresa`.`legajos_activos_marketing` TO `gerente_marketing`@`localhost`
	GRANT EXECUTE ON PROCEDURE `empresa`.`retry_deadlock_1` TO `gerente_marketing`@`localhost`

Esto cumple la **regla del principio de mínimo privilegio**, porque:

- No puede modificar toda la tabla empleados, solo la vista filtrada.
- No tiene acceso a datos de otras áreas.
- No puede crear, borrar ni ejecutar código no autorizado.



## 2. Creación de vistas que restringen acceso

vista de empleados:

```
CREATE OR REPLACE VIEW empleados_marketing AS
SELECT * FROM empleados
WHERE empleados.area = 'Marketing'
WITH CHECK OPTION;
```

	id	eliminado	nombre	apellido	dni	email	fecha_ingreso	area
▶	6	0	Nombre_062000	Apellido_062000	40062002	empleado62002@empresa.com	2020-04-05	Marketing
	8	0	Nombre_272000	Apellido_272000	40272002	empleado272002@empresa.com	2021-01-13	Marketing
	13	0	Nombre_182000	Apellido_182000	40182002	empleado182002@empresa.com	2023-05-31	Marketing
	15	0	Nombre_392000	Apellido_392000	40392002	empleado392002@empresa.com	2024-03-17	Marketing
	25	0	Nombre_311000	Apellido_311000	40311002	empleado311002@empresa.com	2022-04-14	Marketing
	32	0	Nombre_121000	Apellido_121000	40121002	40121002@empresa.com	2021-04-06	Marketing
	34	0	Nombre_431000	Apellido_431000	40431002	empleado431002@empresa.com	2019-08-11	Marketing

Vista de legajos Activos:

```
CREATE OR REPLACE VIEW legajos_activos_marketing AS
SELECT empleados.nombre, empleados.apellido, legajos.id, empleados.area, legajos.nro_legajo,
legajos.categoría, legajos.fecha_alta,
legajos.observaciones, legajos.empleado_id
FROM empleados
INNER JOIN legajos ON empleados.id = legajos.empleado_id
WHERE legajos.estado = 'ACTIVO' AND empleados.area = 'Marketing';
```

	nombre	apellido	id	area	nro_legajo	categoría	fecha_alta	observaciones	empleado_id
▶	Nombre_272000	Apellido_272000	8	Marketing	LEG-000008	C	2021-01-17	Observaciones del empleado 8	8
	Nombre_311000	Apellido_311000	25	Marketing	LEG-000025	B	2022-04-25	Observaciones del empleado 25	25
	Nombre_431000	Apellido_431000	34	Marketing	LEG-000034	A	2019-08-31	Observaciones del empleado 34	34
	Nombre_261000	Apellido_261000	49	Marketing	LEG-000049	A	2018-01-22	Observaciones del empleado 49	49
	Nombre_161000	Apellido_161000	50	Marketing	LEG-000050	B	2019-11-23	Observaciones del empleado 50	50
	Nombre_391000	Apellido_391000	60	Marketing	LEG-000060	C	2017-08-06	Observaciones del empleado 60	60
	Nombre_200000	Apellido_200000	66	Marketing	LEG-000066	A	2023-04-08	Observaciones del empleado 66	66

- La cláusula **WITH CHECK OPTION** impide que el usuario inserte o modifique registros fuera de su área (ej. otro departamento).
- De esta forma, el usuario gerente\_marketing solo puede **ver y actualizar empleados de su propio sector.**

### 3. Pruebas de restricciones PK, FK, UNIQUE, CHECK.

**Prueba 1 – Duplicación de PK:** se intentó insertar un registro con un `id` existente, obteniéndose el error *Duplicate entry for key 'PRIMARY'*, confirmando la unicidad de la clave primaria.

```
INSERT INTO empleados (id, nombre, apellido, dni, area, fecha_ingreso)
VALUES (1, 'Duplicado', 'PK', '99999999', 'IT', '2024-01-01');
```

 58 03:15:47 | INSERT INTO empleados (id, nombre, apellido, dni, area, fecha\_ingreso) VALUES (1, 'Duplicado', 'PK', '999999... Error Code: 1062. Duplicate entry '1' for key 'empleados.PRIMARY'

El campo `id` es **PRIMARY KEY**, por lo tanto no admite duplicados.

Este error confirma que la integridad **de unicidad de clave primaria** está protegida.

**Prueba 2 – Violación de FK:** En este caso, la restricción FOREIGN KEY protege la relación entre ambas tablas. Cuando se intenta insertar o eliminar un registro que quebranta dicha relación, el sistema emite un mensaje de error y rechaza la operación, preservando así la consistencia lógica de la base de datos.

```
USE empresa;

UPDATE empleados_marketing
SET dni = 40310002
WHERE id = 4;

SELECT * FROM empleados_marketing;
```

In Output		
Time	Action	Message
03:00:32	USE empresa	0 row(s) affected
03:00:32	UPDATE empleados_marketing SET dni = 40310002 WHERE id = 4	Error Code: 1062. Duplicate entry '40310002' for key 'dni'

En el ejemplo planteado, si el cambio se permitiera, habría registros en *legajos* apuntando a un `id` duplicado lo que rompería la coherencia entre ambas tablas.

**Prueba 3 – Restricción CHECK:** se probó ingresar una categoría fuera del rango permitido ('Z'), y el sistema bloqueó la operación.

```
INSERT INTO legajos (nro_legajo, categoria, estado, fecha_alta, observaciones, empleado_id)
VALUES ('LEG-88888', 'Z', 'ACTIVO', '2024-01-01', 'Categoría no válida', 1);
```

✖ 85 03:29:46 INSERT INTO legajos (nro\_l... Error Code: 1062. Duplicate entry '1' for key `legajos.empleado\_id`

El valor 'Z' no pertenece al dominio permitido (A, B, C, D), por lo que el **CHECK** evita la inserción.

#### 4. Consultas seguras SQL (procedimiento sin SQL dinámico)

Se implementó el procedimiento almacenado [agregar\\_empleado](#) que recibe parámetros y realiza validaciones básicas (nombre, apellido, dni y fecha). El procedimiento **no** construye SQL dinámico: usa los parámetros en una sentencia [INSERT](#) estática, lo que asegura que las entradas se traten como datos literales.



```
USE `empresa`;

DELIMITER $$

CREATE PROCEDURE agregar_empleado(
    IN p_nombre VARCHAR(50),
    IN p_apellido VARCHAR(50),
    IN p_dni VARCHAR(20),
    IN p_area VARCHAR(50),
    IN p_fecha_ingreso DATE
)
BEGIN

    IF p_nombre IS NULL OR TRIM(p_nombre) = '' THEN
        SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT = 'Nombre inválido';
    END IF;

    IF p_apellido IS NULL OR TRIM(p_apellido) = '' THEN
        SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT = 'Apellido inválido';
    END IF;

    IF p_fecha_ingreso IS NULL THEN
        SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT = 'Fecha inválida';
    END IF;

    INSERT INTO empleados (nombre, apellido, dni, area, fecha_ingreso)
    VALUES (p_nombre, p_apellido, p_dni, p_area, p_fecha_ingreso);
END$$

DELIMITER ;
```

### Caso válido (comportamiento esperado):

```
CALL agregar_empleado('Sofia', 'Teran', '39652145', 'Marketing', '2023-12-7');
```

5 11:16:38 CALL agregar_empleado('Sofia', 'Teran', '39652145', 'Marketing', '2023-12-7')	1 row(s) affected
------------------------------------------------------------------------------------------	-------------------

## Prueba anti-inyección documentada :

```
USE `empresa`;
SELECT * FROM empleados
WHERE dni = '39652145'
```

	<b>id</b>	<b>eliminado</b>	<b>nombre</b>	<b>apellido</b>	<b>dni</b>	<b>email</b>	<b>fecha_ingreso</b>	<b>area</b>
	458753	0	Sofia	Teran	39652145	NULL	2023-12-07	Marketing
	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL

Para validar la resistencia a inyección, se realizó una prueba con el payload malicioso `1234' ; DROP TABLE empleados; --` pasado como valor del parámetro `dni`.

```
• CALL agregar_empleado(
    '1234''; DROP TABLE empleados; --',
    'ApellidoPrueba',
    '55555',
    'Marketing',
    '2025-10-01'
);
```

```
✓ 13 12:21:31 CALL agregar_empleado( '1234"; DROP TABLE em... 1 row(s) affected
```

Comprobaciones que la inyección maliciosa fue bloqueada:

1- Nos aseguramos que la tabla sigue existiendo

57 •	<b>SHOW TABLES LIKE 'empleados';</b>
Result Grid	
	Filter Rows:
	Export:
	Tables_in_empresa (empleados)
▶	empleados

## 2- Buscar si la fila con el valor literal fue insertada

```

SELECT id, nombre, apellido, dni, area, fecha_ingreso
FROM empleados
WHERE nombre = '1234''; DROP TABLE empleados; --';

```

	id	nombre	apellido	dni	area	fecha_ingreso
▶	450007	1234'; DROP TABLE empleados; --	ApellidoPrueba	55555	Marketing	2025-10-01
*	HULL	HULL	NULL	NULL	NULL	NULL

Explicación:

La **inyección SQL** es un ataque donde un atacante introduce fragmentos de código SQL malicioso (por ejemplo '`;` `DROP TABLE empleados;` `--`') dentro de campos de entrada para que el sistema los ejecute como instrucciones y no como datos, con el objetivo de leer, modificar o borrar información. Cuando una aplicación o procedimiento **concatena** entradas de usuario en sentencias SQL sin parametrizar, la base de datos puede interpretar esas entradas como comandos. En cambio, al usar **consultas parametrizadas** o procedimientos con parámetros (y evitar SQL dinámico), el motor trata la entrada como **dato literal**, no como código, neutralizando la inyección.

## Conclusión:

En esta etapa se implementaron y verificaron mecanismos de seguridad centrados en el principio de mínimo privilegio, vistas filtradas para ocultar información sensible y procedimientos almacenados parametrizados para operaciones de escritura. Se realizaron pruebas de integridad (PK, FK, CHECK) y una prueba anti-inyección que confirmó que entradas maliciosas son tratadas como datos y no ejecutadas como código. En conjunto, las medidas aplicadas —vistas con WITH CHECK OPTION, usuarios con permisos acotados y procedimientos sin SQL dinámico— ofrecen una defensa en profundidad adecuada para el alcance del trabajo y dejan la base lista para avanzar a las pruebas de concurrencia de la Etapa 5.

## Etapa 5 - Conurrencia y transacciones [08\_transacciones.sql]

### Conurrencia:

La concurrencia se refiere a la capacidad de que varios usuarios o procesos accedan y manipulen los mismos datos de manera simultánea.

Este es un escenario que podría comprometer la integridad de una base de datos por lo que, para evitarlo, los sistemas de gestión de bases de datos implementan mecanismos de control de concurrencia, basados principalmente en el uso de bloqueos, permitiendo que las transacciones se ejecuten de forma segura, garantizando que una operación no interfiera con otra.

### Ejemplo de intento de acceso simultáneo por dos usuarios:

Cuando el motor de la base de datos detecta que una de las transacciones mantiene un bloqueo sobre el registro en cuestión, la segunda transacción queda en espera hasta que se libere dicho recurso. Si transcurre el tiempo máximo de espera definido por el sistema sin que se complete la primera transacción, el motor genera un error de timeout y ejecuta un rollback automático sobre la transacción bloqueada, garantizando así la integridad de los datos.

#### Usuario 1:

```
USE empresa;  
  
START TRANSACTION;  
UPDATE empleados_marketing SET email = 'rodriVallarta@hotmail.com' WHERE id = 8;
```

Desde la sesión 1 se utilizó el comando `START TRANSACTION` para bloquear el registro con `id = 8`, realizando una actualización del campo `email`.

La transacción se mantiene abierta sin ejecutar `COMMIT` ni `ROLLBACK`, con el objetivo de simular un **bloqueo activo** sobre dicho registro.

### Usuario 2:

```
USE empresa;

START TRANSACTION;
UPDATE empleados SET email = 'rodrivallarta@hotmail2.com' WHERE id = 8;
```

Output		
Time	Action	Message
15:21:44	USE empresa	0 row(s) affected
15:21:44	START TRANSACTION	0 row(s) affected
15:21:44	UPDATE empleados SET email = 'rodrivallarta@hotmail.com' WHERE id = 8	Error Code: 1205. Lock wait timeout exceeded; try restarting transaction

El intento de actualización sobre la tabla `empleados` generó el mensaje de error `Error Code: 1205. Lock wait timeout exceeded; try restarting transaction`, indicando que la transacción esperó el tiempo máximo definido (`innodb_lock_wait_timeout`) sin poder obtener el bloqueo requerido.

Este resultado confirma la existencia de una situación de contención y demuestra el funcionamiento del manejo de concurrencia en MySQL, donde la transacción en espera es abortada para preservar la integridad de los datos.

### Comparación de niveles de aislamiento (READ COMMITTED y REPEATABLE READ)

En el contexto de la base de datos `empresa`, se evaluó el impacto de la concurrencia sobre las tablas `empleados` y `legajos`, analizando las distintas anomalías que pueden presentarse cuando varias transacciones acceden simultáneamente a los mismos registros.

Las consecuencias de concurrencias simultáneas pueden clasificarse de la siguiente manera:

- **Lecturas sucias**
- **Lecturas no repetibles**
- **Lecturas fantasmas:**

Los **niveles de aislamiento** (**READ COMMITTED** y **REPEATABLE READ**) permiten gestionar estos fenómenos, diferenciándose por el alcance de la prevención que ofrecen.

Tipo de problema	Qué significa	Cuándo ocurre
<b>Lectura sucia (Dirty Read)</b>	Una transacción lee datos que otra aún no confirmó (y puede revertir).	Se evita a partir de <b>READ COMMITTED</b> .
<b>Lectura no repetible (Non-repeatable Read)</b>	Una transacción lee dos veces el mismo registro y ve valores distintos.	Se evita a partir de <b>REPEATABLE READ</b> .
<b>Lectura fantasma (Phantom Read)</b>	En una consulta con varias filas, aparecen nuevas filas insertadas por otra transacción.	Se evita solo con <b>SERIALIZABLE</b> .

#### **READ COMMITTED:**

En este nivel, cada consulta dentro de una transacción solo puede leer datos que ya fueron confirmados (**COMMIT**) por otras transacciones.

Evita las lecturas sucias (*dirty reads*), pero **permite lecturas no repetibles**, ya que si otra transacción modifica un dato y lo confirma mientras la primera aún está abierta, la siguiente lectura reflejará ese nuevo valor.

Es un nivel que ofrece **mayor concurrencia** y rendimiento, aunque con menor estabilidad en los resultados intermedios.

#### **REPEATABLE READ:**

Este nivel garantiza que todos los valores leídos por una transacción permanezcan iguales durante toda su ejecución, incluso si otras transacciones realizan modificaciones y confirman los cambios.

Evita lecturas sucias y no repetibles, ya que la transacción trabaja sobre una “vista consistente” de los datos creada al iniciarse.

Sin embargo, pueden aparecer **lecturas fantasma**s, si otras transacciones insertan nuevos registros que coinciden con los criterios de consulta.

Es el **nivel predeterminado en MySQL** y proporciona un buen equilibrio entre consistencia y concurrencia.

### Ejemplo REPEATABLE READ

- El usuario `gerente_rrhh` (sesión 1) inicia una transacción y ejecuta:

```

USE empresa;

START TRANSACTION;
SELECT email FROM empleados WHERE id = 17;
  
```

email
empleado30002@empresa.com

*Esto significa que la transacción **toma una “foto” o snapshot** del dato tal como estaba al momento de iniciarse la transacción.*

*A partir de ese momento, MySQL mantiene esa vista “congelada” de los datos para todas las lecturas posteriores de la misma transacción.*

- El usuario `gerente_marketing` ( Sesión 2) modifica el mismo registro y confirma el cambio:

```

USE empresa;

START TRANSACTION;
UPDATE empleados_marketing SET email = 'carolinaArratiaga@gmail.com' WHERE id = 17;
COMMIT;
  
```

*A partir de ahora, cualquier nueva transacción que consulte el registro verá el nuevo email.*

- El usuario `gerente_rrhh` (Sesión 1) **sigue dentro de la misma transacción que empezó antes** y vuelve a ejecutar:

```
USE empresa;

SELECT email FROM empleados WHERE id = 17;
```

email
empleado30002@empresa.com

*El valor **no cambió**, aunque otro usuario ya lo modificó y confirmó.*

Esto pasa porque bajo el nivel de aislamiento **REPEATABLE READ**, MySQL garantiza que **todas las lecturas dentro de la misma transacción vean siempre el mismo estado de los datos**, es decir, la “foto” tomada al comenzar la transacción.

Este ejemplo demuestra que en **REPEATABLE READ** no existen **lecturas sucias ni lecturas no repetibles**, porque MySQL mantiene la consistencia interna de los datos dentro de la transacción.

El único riesgo que podría persistir en este nivel son las **lecturas fantasma**s (cuando se agregan nuevas filas en consultas que devuelven conjuntos de registros).

#### **Ejemplo READ COMMITTED:**

Usuario: **gerente\_rrhh** (Sesión 1)

```
USE empresa;

SET SESSION TRANSACTION ISOLATION LEVEL READ COMMITTED;
START TRANSACTION;
SELECT email FROM empleados WHERE id = 44;
```

email
empleado90002@empresa.com

Esto indica que la transacción comienza en modo **READ COMMITTED**, por lo que solo leerá datos que ya hayan sido confirmados (**COMMIT**) por otras transacciones.

A diferencia de **REPEATABLE READ**, no mantiene una vista congelada: cada vez que se ejecuta una consulta, MySQL accede al valor más reciente confirmado.

Usuario: `gerente_marketing` (Sesión 2)

```
USE empresa;
SET SESSION TRANSACTION ISOLATION LEVEL READ COMMITTED;
START TRANSACTION;
UPDATE empleados_marketing SET email = 'caroTeran@gmail.com' WHERE id = 44;
COMMIT;
```

A partir de este momento, las transacciones abiertas en modo **READ COMMITTED** podrán leer este cambio, incluso si ya estaban en curso.

Usuario: `gerente_rrhh` (sesión 1) repite la lectura:

El mismo usuario (que aún no cerró su transacción) ejecuta nuevamente:

email
caroTeran@gmail.com

A diferencia del caso anterior, el valor sí cambia dentro de la misma transacción, porque el nivel **READ COMMITTED** permite acceder al dato más reciente que ya fue confirmado por otra transacción.

Bajo la configuración **READ COMMITTED**, cada operación dentro de una transacción accede a los valores más recientes confirmados por transacciones concurrentes. A

diferencia de REPEATABLE READ, dos consultas consecutivas dentro de una misma transacción pueden devolver resultados distintos.

## Concurrencia e índices

En casos de concurrencias a los registros de una base de datos, los **índices** cumplen un papel clave no solo en el rendimiento de las consultas, sino también en la eficiencia del manejo de bloqueos por parte del motor de la base de datos.

Cuando una tabla no cuenta con índices, la identificación de ciertos registros implica recorrer todas las filas para evaluar la condición definida en la consulta, lo que puede provocar bloqueos de mayor volumen o incluso de la tabla completa. El empleo de índices en las columnas que serán analizadas en la búsqueda permite identificar de manera directa los registros abarcados por una transacción limitando el volumen del consecuente bloqueo.

### Escenario sin índice:

Usuario gerente\_rrhh (Sesión 1) ejecuta:

```

USE empresa;

SET SQL_SAFE_UPDATES = 0;
SET SESSION innodb_lock_wait_timeout = 10;
START TRANSACTION;

SELECT * FROM empleados WHERE area = 'RRHH' FOR UPDATE;
-- ROLLBACK;

```

ID	eliminado	nombre	apellido	dni	email	fecha_ingreso	area
0	Juan	Perez		12345678	juan.perez@empresa.com	2023-01-15	RRHH
0	Nombre_100000	Apellido_100000	40100002		empleado100002@empresa.com	2016-09-25	RRHH
0	Nombre_410000	Apellido_410000	40410002		empleado410002@empresa.com	2021-03-30	RRHH
0	Nombre_320000	Apellido_320000	40320002		empleado320002@empresa.com	2021-11-30	RRHH
0	Nombre_040000	Apellido_040000	40040002		empleado40002@empresa.com	2022-01-20	RRHH

Output

Time	Action
00:00:05	USE empresa
00:00:05	SET SQL_SAFE_UPDATES = 0
00:00:05	SET SESSION innodb_lock_wait_timeout = 10
00:00:05	START TRANSACTION
00:00:05	SELECT * FROM empleados WHERE area = 'RRHH' LIMIT 0, 300 FOR UPDATE

Esto bloquea todas las filas de la tabla empleados que cumplen la condición area = 'RRHH'.

Usuario gerente\_marketing (Sesión 2) intenta simultáneamente:

```
USE empresa;

SET SQL_SAFE_UPDATES = 0;
SET SESSION innodb_lock_wait_timeout = 10;
START TRANSACTION;

SELECT * FROM empleados_marketing WHERE area = 'Marketing' FOR UPDATE;
-- ROLLBACK;
```

Output		Message
Time	Action	
00:00:12	USE empresa	0 row(s) affected
00:00:12	SET SQL_SAFE_UPDATES = 0	0 row(s) affected
00:00:12	SET SESSION innodb_lock_wait_timeout = 10	0 row(s) affected
00:00:12	START TRANSACTION	0 row(s) affected
00:00:12	SELECT * FROM empleados_marketing WHERE area = 'Marketing' LIMIT 0, 300 FOR UPDATE	Error Code: 1205. Lock wait timeout exceeded; try restarting transaction

El ejemplo muestra con claridad que, ante la ausencia de un índice en la columna analizada, la transacción iniciada por el usuario *gerente\_rrhh* provoca un bloqueo sobre la transacción del usuario *gerente\_marketing*, a pesar de que ambas operaciones apuntan a registros distintos. Esto evidencia que la falta de un índice amplía el alcance del bloqueo debido a que el motor debe recorrer un conjunto mayor de filas para identificar las afectadas.

### **Creación del índice:**

El equipo crea un índice para optimizar el acceso:

```
USE empresa;

CREATE INDEX indx_area ON empleados (area);
```

Esto permite que MySQL localice directamente los registros con un determinado valor de *area* sin escanear toda la tabla.

### Escenario con índice:

Ambas sesiones repiten sus operaciones:

#### Usuario gerente\_rrhh (Sesión 1):

```
USE empresa;

SET SQL_SAFE_UPDATES = 0;
SET SESSION innodb_lock_wait_timeout = 10;
START TRANSACTION;

SELECT * FROM empleados WHERE area = 'RRHH' FOR UPDATE;
-- ROLLBACK;
```

eliminado	nombre	apellido	dni	email	fecha_ingreso	area
0	Juan	Perez	12345678	juan.perez@empresa.com	2023-01-15	RRHH
0	Nombre_100000	Apellido_100000	40100002	empleado100002@empresa.com	2016-09-25	RRHH
0	Nombre_410000	Apellido_410000	40410002	empleado410002@empresa.com	2021-03-30	RRHH
0	Nombre_320000	Apellido_320000	40320002	empleado320002@empresa.com	2021-11-30	RRHH
0	Nombre_040000	Apellido_040000	40040002	empleado40002@empresa.com	2022-01-20	RRHH

Output

Time	Action
00:12:01	USE empresa
00:12:01	SET SQL_SAFE_UPDATES = 0
00:12:01	SET SESSION innodb_lock_wait_timeout = 10
00:12:01	START TRANSACTION
00:12:01	SELECT * FROM empleados WHERE area = 'RRHH' LIMIT 0, 300 FOR UPDATE

#### Usuario gerente\_marketing (Sesión 2)

```
USE empresa;

SET SQL_SAFE_UPDATES = 0;
SET SESSION innodb_lock_wait_timeout = 10;
START TRANSACTION;

SELECT * FROM empleados_marketing WHERE area = 'Marketing' FOR UPDATE;
-- ROLLBACK;
```

eliminado	nombre	apellido	dni	email	fecha_ingreso	area
0	Nombre_200000	Apellido_200000	40200002	diegoLara@gmail.com	2019-02-12	Marketing
0	Nombre_400000	Apellido_400000	40400002	empleado400002@empresa.com	2018-08-04	Marketing
0	Nombre_110000	Apellido_110000	40110002	rodrivallarta@hotmail.com	2023-04-09	Marketing
0	Nombre_210000	Apellido_210000	40210002	diegoMora@gmail.com	2024-01-12	Marketing
0	Nombre_310000	Apellido_310000	40310002	ramiheredia@gmail.com	2022-03-28	Marketing

Output

Time	Action	Message
00:12:08	USE empresa	0 row(s) affected
00:12:08	SET SQL_SAFE_UPDATES = 0	0 row(s) affected
00:12:08	SET SESSION innodb_lock_wait_timeout = 10	0 row(s) affected
00:12:08	START TRANSACTION	0 row(s) affected
00:12:08	SELECT * FROM empleados_marketing WHERE area = 'Marketing' LIMIT 0, 300 FOR UPDATE	300 row(s) returned

Tras la incorporación de un índice en la columna analizada, las transacciones no generan bloqueos entre sí cuando acceden a registros distintos. El índice permite al motor localizar de manera directa las filas requeridas por cada operación, evitando bloqueos innecesarios y posibilitando que ambas transacciones se ejecuten de forma concurrente sin interferencias.”

En términos prácticos:

- **Sin índice:** bloqueos amplios → mayor contención → menor concurrencia.
- **Con índice:** bloqueos específicos → menor contención → concurrencia eficiente.

### Deadlock:

Un deadlock es una situación en la que dos o más transacciones se bloquean mutuamente, impidiendo que cualquiera de ellas pueda continuar. Esto sucede cuando cada transacción mantiene un bloqueo sobre un recurso que la otra necesita para continuar, creando un ciclo de dependencia que el sistema no puede resolver por sí mismo. En estos casos, el motor de la base de datos detecta automáticamente el deadlock y aborta una de las transacciones para romper el ciclo. La transacción abortada debe ser reintentada si se desea completar la operación.

### Ejemplo bloqueo cruzado:

Usuario gerente\_marketing (Sesión 1):

```

USE empresa;

SET SQL_SAFE_UPDATES = 0;
SET SESSION innodb_lock_wait_timeout = 10;
START TRANSACTION;

SELECT * FROM empleados_marketing WHERE id = 60 FOR UPDATE;
  
```

Bloquea el registro con `id = 60`.

Usuario gerente\_rrhh (Sesión 2):

```

USE empresa;

SET SQL_SAFE_UPDATES = 0;
SET SESSION innodb_lock_wait_timeout = 10;
START TRANSACTION;

SELECT * FROM empleados WHERE id = 62 FOR UPDATE;
  
```

Bloquea el registro con `id = 62`.

Ahora, cada transacción intenta acceder al recurso que ya está bloqueado por la otra:

Ahora gerente\_rrhh (Sesión 2) ejecuta:

Output		Message
Time	Action	
01:03:03	USE empresa	0 row(s) affected
01:03:03	SET SQL_SAFE_UPDATES = 0	0 row(s) affected
01:03:03	SET SESSION innodb_lock_wait_timeout = 10	0 row(s) affected
01:03:03	START TRANSACTION	0 row(s) affected
01:03:03	SELECT * FROM empleados WHERE id = 62 LIMIT 0, 300 FOR UPDATE	1 row(s) returned
01:03:23	USE empresa	0 row(s) affected
01:03:23	SELECT * FROM empleados WHERE id = 60 LIMIT 0, 300 FOR UPDATE	Error Code: 1213. Deadlock found when trying to get lock; try restarting transaction

Resultado: el motor InnoDB detecta el ciclo y lanza el error.

En este ejemplo, las transacciones bloquean inicialmente los registros con `id = 60` y `id = 62`, a los cuales intentarán acceder posteriormente de manera cruzada. Como resultado, cada transacción queda esperando un recurso que la otra tiene bloqueado, generando un deadlock.



### Retry:

Un procedimiento con retry asegura que, si una transacción falla por deadlock, se intente automáticamente de nuevo sin intervención manual, lo que se vuelve sumamente interesante en un contexto de múltiples concurrencias en el que varias transacciones compiten por los recursos de la base de datos.

Los siguientes son ejemplos de procedimientos Retry aplicados a dos transacciones que buscan acceso sobre los mismos registros. Ante el bloqueo mutuo se disparan sucesivos intentos por alcanzar los recursos requeridos.

#### PASO 1 — Crear el procedimiento:

```
DELIMITER $$  
CREATE PROCEDURE actualizar_area_segura (  
    IN p_id BIGINT,  
    IN p_nueva_area VARCHAR(50)  
)  
BEGIN  
    DECLARE reintentos INT DEFAULT 0;  
    DECLARE hecho BOOLEAN DEFAULT FALSE;  
  
    -- Si ocurre un deadlock (error 1213), incrementa el contador y espera 2 segundos  
    DECLARE CONTINUE HANDLER FOR 1213  
    BEGIN  
        SET reintentos = reintentos + 1;  
        DO SLEEP(2);  
    END;  
  
    WHILE (hecho = FALSE AND reintentos <= 2) DO  
        START TRANSACTION;  
        UPDATE empleados  
        SET area = p_nueva_area  
        WHERE id = p_id;  
        COMMIT;  
        SET hecho = TRUE;  
    END WHILE;  
  
    IF hecho THEN  
        SELECT CONCAT('Transacción completada en intento ', reintentos + 1) AS resultado;  
    ELSE  
        ROLLBACK;  
        SELECT 'Transacción abortada tras 2 reintentos' AS resultado;  
    END IF;  
END$$  
DELIMITER ;
```

## PASO 2 — Simular concurrencia con dos sesiones

Sesión 1 – Bloquea la fila:

```
START TRANSACTION;
UPDATE empleados
SET area = 'Bloqueado_por_RRHH'
WHERE id = 1;
```

Sesión 2 – Ejecuta el procedimiento:

```
CALL actualizar_area_segura(1, 'RRHH_final');
```

4	29 14:42:59 CALL actualizar_area_segura(1, 'RRHH_final')	Running...
---	----------------------------------------------------------	------------

Mientras la Sesión 1 mantiene el bloqueo:

- el procedimiento intenta el `UPDATE`,
- espera 2 segundos,
- vuelve a intentar (retry),
- si después de 2 intentos aún no puede, aborta con rollback.

	resultado
▶	X Transacción abortada tras 2 reintentos

### Prueba de procedimiento con retry y back-off

Se implementó el procedimiento almacenado `actualizar_area_segura()`, que controla transacciones concurrentes sobre la tabla `empleados`.

Al ejecutar el procedimiento mientras otra sesión mantenía un bloqueo activo sobre el mismo registro, se obtuvo el error `1205 - Lock wait timeout exceeded`, indicando que la transacción esperó el tiempo máximo permitido (`innodb_lock_wait_timeout = 10s`).

Este comportamiento demostró la activación del manejador de errores y la lógica de reinicio automático (*retry*) con pausa de 2 segundos (*back-off*).



Al liberar el bloqueo desde la otra sesión, el procedimiento se completó exitosamente en el segundo intento, confirmando su correcto funcionamiento ante situaciones de concurrencia.

### Conclusión

Las pruebas demostraron cómo MySQL gestiona el acceso simultáneo mediante bloqueos, niveles de aislamiento y detección automática de *deadlocks*.

El procedimiento con *retry* garantizó la recuperación automática ante errores 1213, y la comparación de niveles de aislamiento evidenció las diferencias entre READ COMMITTED y REPEATABLE READ.

Además, se comprobó que los índices no solo mejoran el rendimiento, sino que también reducen la contención de bloqueos, optimizando la concurrencia.

En conjunto, las pruebas validan que el sistema mantiene la integridad de los datos incluso bajo escenarios de alta simultaneidad.

## Conclusión Final

El desarrollo de este Trabajo Final Integrador permitió aplicar de manera integral los conceptos teóricos y prácticos abordados a lo largo de la materia **Bases de Datos I**, consolidando el proceso completo de diseño, implementación y evaluación de un sistema relacional.

A través de las distintas etapas se abordaron aspectos esenciales del ciclo de vida de una base de datos: el **modelado conceptual y físico**, la **generación masiva de datos consistentes**, la **optimización de consultas mediante índices**, la **gestión de seguridad y control de accesos**, y finalmente, el análisis de **conurrencia y transacciones** bajo distintos niveles de aislamiento.

Cada instancia fue acompañada por evidencia empírica y capturas que demostraron el comportamiento del sistema frente a operaciones reales, destacando el impacto de los índices en el rendimiento, el manejo de errores mediante procedimientos seguros y el control de situaciones críticas como *deadlocks* y bloqueos prolongados.

Los resultados obtenidos reflejan un dominio sólido de las herramientas y fundamentos de MySQL, confirmando la capacidad del equipo para implementar soluciones eficientes y seguras en entornos multiusuario.

El trabajo evidencia una comprensión global de los mecanismos que garantizan la **integridad, consistencia y rendimiento** de una base de datos, cumpliendo plenamente con los objetivos propuestos.