

Tecnológico Nacional de México
Instituto Tecnológico de Mexicali

Ingeniería en Sistemas Computacionales



Taller de Base de Datos

Reporte de práctica. Prácticas 4.1 a 4.2 y 5.1 a 5.7.

Barba Navarro Luis Rodrigo, 20490687
Gurrola Bernal Johan Antonio, 20490703
Lira López Alejandro, 16491160
Pérez García Sofía, 20491083

No. Equipo: 4
Grupo: 3 p.m - 4 p.m, CZA 5
Profesor: Carlos Alberto López Castellanos

Mexicali, Baja California a domingo, 11 de diciembre de 2022.

Índice.

I. Introducción.	1
II. Justificación.	3
III. Objetivos.	4
IV. Desarrollo.	5
A. Teoría preliminar sobre procedimientos almacenados, manejadores y disparadores.	5
B. Teoría Preliminar sobre transacciones.	9
C. Práctica 4.1.	12
D. Ejemplo de la Práctica 4.1 con base de datos ‘maquina_expendedora’.	18
E. Práctica 4.2.	23
F. Ejemplo de la Práctica 4.2 con base de datos ‘maquina_expendedora’.	32
G. Práctica 5.1 sobre procedimientos almacenados.	36
H. Ejemplo de la práctica 5.1 con base de datos ‘maquina_expendedora’.	37
I. Práctica 5.2 sobre procedimientos almacenados con variables.	38
J. Ejemplo de la práctica 5.2 con base de datos ‘maquina_expendedora’.	41
K. Práctica 5.3 sobre procedimientos almacenados con parámetros de entrada y salida.	43
L. Ejemplo de la práctica 5.3 con base de datos ‘maquina_expendedora’.	44
M. Práctica 5.4 sobre procedimientos almacenados con estructuras de control.	45
N. Ejemplo de la práctica 5.4 con base de datos ‘maquina_expendedora’.	51
O. Práctica 5.5 sobre manejadores en procedimientos almacenados.	58
P. Ejemplo de la práctica 5.5 con base de datos ‘maquina_expendedora’.	60
Q. Práctica 5.6 sobre disparadores en procedimientos almacenados.	63
R. Ejemplo de la práctica 5.6 con base de datos ‘maquina_expendedora’.	68
S. Práctica 5.7 sobre disparadores en procedimientos almacenados con conceptos mixtos.	71
T. Ejemplo de la práctica 5.7 con base de datos ‘maquina_expendedora’.	75
V. Conclusión y recomendaciones.	77

I. Introducción.

En el momento en el que nos toca administrar una base de datos, se pueden ejecutar varias operaciones que pueden necesitar ser rastreadas en caso de errores o particularmente por razones de seguridad.

Por ejemplo, cuando se desea retirar dinero en un cajero automático, en este caso nosotros ingresamos la cantidad que se desea retirar y el sistema verifica en la base de datos para determinar si puede retirar la cantidad solicitada. Si es así, el sistema deduce el dinero de la base de datos y le envía un mensaje de éxito.

Sin embargo, si llegara haber un corte de energía o algún problema de conexión que afecte el proceso de transacción, el proceso de retiro puede fallar mientras está en progreso. Las transacciones nos ayudan a realizar un seguimiento de cualquier modificación realizada en la base de datos y manejarla con mayor precisión. Para casos como este, los cambios se pueden revertir o sobrescribir. El manejo de transacciones a nivel empresarial tiene gran relevancia para básicamente garantizar que la base de datos nunca contenga el resultado de operaciones parciales. En un conjunto de operaciones, si una de ellas falla, se produce la reversión para restaurar la base de datos a su estado original.

En otro contexto de la cotidianidad, es importante tener como programador buenas técnicas al momento de codificar. A medida que nuestro código va evolucionando, es mejor llevar a cabo mejores prácticas que permitan que el código sea más fácil de entender, también que sea eficiente y productivo y evite el caso de sobreponer código o cosas que perjudiquen la integridad del programa. En el contexto de las bases de datos, es totalmente común mencionar lo relacionado a los procedimientos almacenados, los cuales nos proporcionan una importante capa de seguridad entre la interfaz de usuario y la base de datos. A su vez, nos permite mejorar la seguridad de nuestra base de datos ya que se basa a través de controles de acceso a datos, donde los usuarios finales pueden ingresar o modificar datos, pero no escribir o modificar procedimientos.

Por otro lado, por cuestiones del lado del administrador de la base de datos o por parte del usuario, al momento de manejar procedimientos almacenados cabe la posibilidad de que se generen errores; si se produce un error cuando se ejecuta un procedimiento, el procedimiento finaliza a menos que incluya instrucciones para indicarle al procedimiento que realice alguna otra acción. Estas declaraciones se denominan manejadores. Básicamente nos indica un después cuando algo falla en nuestro programa; nos benefician en el sentido de controlar los posibles errores que se pueden generar a futuro y cómo maniobrar las acciones.

Por lo general, en las empresas se busca la mejora continua, la calidad y automatización de los procesos, sobre todo refiriéndonos a las bases de datos, yace el término de disparador, donde su objetivo principal es automatizar la ejecución de código cuando ocurre un evento dentro de nuestra base de datos. En palabras más simples, si se necesita que un determinado bloque de código se ejecute siempre en respuesta a un evento determinado, hacer uso de disparadores es nuestra mejor opción. Principalmente porque garantizan que el código se ejecutará o que el evento que disparó el disparador fallará.

En el presente reporte de práctica, se pretenden realizar las prácticas aplicadas al principio de la unidad, que tienen relación con la aplicación de transacciones en la base de datos 'mydb' y la base de datos del proyecto 'maquina_expendedora'. Donde se observará y se explicará su comportamiento y se verificará qué implicaciones tiene sobre el total de registros de una tabla. Asimismo, se pretende solucionar algunos de los problemas relacionados a procedimientos almacenados, manejadores y disparadores, así como una implementación mixta de estos conceptos. Asimismo, se pretende manifestar algunos ejemplos de prácticas planteadas con respecto a la base de datos del proyecto en cuestión de las necesidades de una máquina expendedora.

II. Justificación.

La razón principal por la cual se realizó esta práctica fue principalmente para aumentar mi conocimiento relacionado al manejo de transacciones dentro de una base de datos, además, para reconocer la importancia que tienen las transacciones

para asegurar la integridad de los datos almacenados al momento de querer realizar una modificación y que en el proceso surja algún tipo de desacuerdo.

Por otra parte, otra de las razones por las cuales es recomendable practicar el manejo de las transacciones es debido a que la mayoría de sistemas a nivel global ya adoptan las transacciones como soluciones de bases de datos en métodos de operación; es importante familiarizarnos con la implementación para que nuestros sistemas sean flexibles y seguros.

Otra de las razones por la cual se pretende realizar este reporte de práctica es para aumentar el conocimiento relacionado a los procedimientos almacenados, manejadores, disparadores y en casos generales, la implementación de los conceptos mencionados de manera mixta. En la actualidad, las empresas que tratan de gestionar el almacenamiento de información, buscan personal capacitado sobre administración de base de datos, ya que estos les permiten como empresa guiar hacia un manejo de grandes volúmenes de información. Como administradores de base de datos, se tratará de buscar las opciones más viables y óptimas que permitan hacer lo que se requiere en un intervalo de tiempo menor, y permitir que las pérdidas económicas se reduzcan.

Por otro lado, continuando con las razones fundamentales, por la cual se optó por realizar este reporte de práctica fue para concientizar sobre las buenas prácticas del administrador de base de datos, ya que manejar conceptos tan amplios puede llegar a ser agotador y desgastante más a medida que el programa va escalando, pero se trata de la única manera de llevar un control nuestro código; aportando bloques de código que puedan ser reutilizados a futuro, y tratar en la medida de lo posible que se reduzcan los errores.

III. Objetivos.

- A. Controlar la concurrencia de la base de datos, para disminuir los problemas de desempeño y/o consistencia
- B. Implementar transacciones dentro de una base de datos para salvaguardar la integridad de la información.

- C. Observar el comportamiento que tienen determinadas transacciones en el proceso de ejecución dentro de una base de datos.
- D. Conocer la teoría preliminar que involucran los procedimientos almacenados, manejadores y disparadores, así como su implementación de manera mixta.
- E. Buscar soluciones que permitan desarrollar los procedimientos almacenados, manejadores y disparadores, así como su implementación de manera mixta.
- F. Analizar los resultados obtenidos por parte de las prácticas y brindar una explicación objetiva de lo que se presentó.
- G. Reconocer la importancia de los procedimientos almacenados, manejadores y disparadores, así como su implementación de manera mixta sobre todo en el contexto empresarial.

IV. Desarrollo.

A. Teoría preliminar sobre procedimientos almacenados, manejadores y disparadores.

Procedimientos almacenados.

¿Qué es un procedimiento almacenado en SQL?

Un procedimiento almacenado en SQL es una unidad reutilizable que encapsula la lógica empresarial específica de la aplicación. Un procedimiento SQL es un grupo de sentencias SQL y lógica, compiladas y almacenadas juntas para realizar una tarea específica.

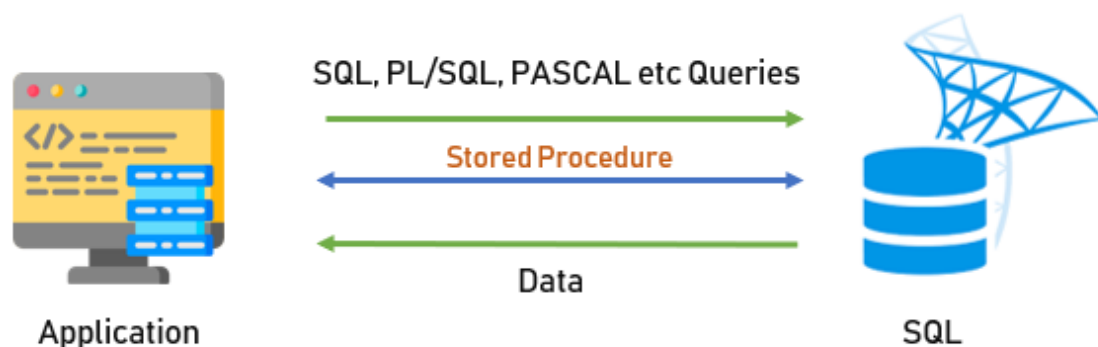


Figura 1.1. Representación gráfica de un procedimiento almacenado.

Características de un procedimiento almacenado en SQL.

- Fácil de implementar porque utilizan un lenguaje muy simple de alto nivel y fuertemente tipado

- Admite tres tipos de parámetros, a saber, parámetros de entrada, salida y entrada-salida.
- Más fiable que los procedimientos externos equivalentes.
- Los procedimientos SQL promueven la reutilización y la mantenibilidad.
- Admite un modelo simple pero potente de control de errores y condiciones.
- Devolver un valor de estado a un procedimiento o lote de llamada para indicar el éxito o el error y el motivo del error.

Sintaxis de procedimientos almacenados en SQL.

Ahora que sabe qué son los procedimientos, características y por qué son necesarios dentro de nuestras bases de datos, se analizará la sintaxis y código de implementación de un procedimiento almacenado en SQL.

Ejemplo de sintaxis de procedimiento almacenado en SQL.

```
CREATE [OR REPLACE] PROCEDURE procedure_name [
(parameter_name [IN | OUT | IN OUT] type [ ])]
{IS | AS }
BEGIN [declaration_section]
executable_section
//SQL statement used in the stored procedure
END
GO
```

Terminologías de sintaxis.

Parámetro.

Un parámetro es una variable que contiene un valor de cualquier tipo de datos SQL válido a través del cual el subprograma puede intercambiar los valores con el código principal. En otras palabras, los parámetros se utilizan para pasar valores al procedimiento. Hay 3 tipos diferentes de parámetros, que son los siguientes:

- IN: Es el parámetro predeterminado, que siempre recibe los valores del programa que llama. Es una variable de sólo lectura dentro de los subprogramas y su valor no se puede cambiar dentro del subprograma.
- OUT: Se utiliza para obtener resultados de los subprogramas.

- INOUT: Este parámetro se utiliza tanto para dar entrada como para obtener salida de los subprogramas.

Otras terminologías.

- *procedure-name* especifica el nombre del procedimiento. Debe ser único.
 - La opción *[OR REPLACE]* permite la modificación de un procedimiento existente.
 - *IS | AS*, establecen el contexto para ejecutar el procedimiento almacenado. La diferencia es que la palabra clave 'IS' se usa cuando el procedimiento está anidado en otros bloques y si el procedimiento es independiente, se usa 'AS'.
 - *code_block* declara las instrucciones de procedimiento que controlan todo el procesamiento dentro del procedimiento almacenado.
-

Manejadores.

¿Qué son los manejadores?

Si se produce un error cuando se ejecuta un procedimiento SQL, el procedimiento finaliza a menos que incluya instrucciones para indicar el procedimiento para realizar alguna otra acción. Estas instrucciones se denominan manejadores.

Los manejadores son similares a las instrucciones WHENEVER en la aplicación SQL externa Programas. Los controladores indican al procedimiento SQL qué hacer cuando se produce un error o se produce una advertencia, o cuando no se devuelven más filas de una consulta. Además, puede declarar controladores para SQLSTATE específicos. Tú puede hacer referencia a un SQLSTATE por su número en un controlador, o puede declarar un nombre para SQLSTATE y, a continuación, use ese nombre en el manejador.

Sintaxis de los manejadores.

La forma general de una declaración de manejador es:

```
DECLARE handler-type HANDLER FOR condition
      SQL-procedure-statement;
```


En general, la forma en que funciona un controlador es cuando se produce un error que coincide con la condición.

Tipos de manejadores.

El tipo de manejador determina lo que sucede después de completar la instrucción del procedimiento. Puede declarar que el tipo de controlador es CONTINUE o EXIT.

- **CONTINUE:** Especifica que después de que se complete la instrucción de procedimiento, la ejecución continúa con la instrucción después de la instrucción que causó el error.
- **EXIT:** Especifica que una vez completada la instrucción de procedimiento, la ejecución continúa al final de la instrucción compuesta que contiene el manejador.

Disparadores.

¿Qué es un disparador?

Los disparadores son objetos de base de datos, en realidad, un tipo especial de procedimiento almacenado, que "reacciona" a ciertas acciones que realizamos en la base de datos. La idea principal detrás de los disparadores es que siempre realizan una acción en caso de que ocurra algún evento.

Tipos de disparadores.

1. **Disparadores DML** (lenguaje de manipulación de datos): reaccionan a los comandos DML. Estos son: INSERT, UPDATE y DELETE
2. **Disparadores DDL** (lenguaje de definición de datos): como era de esperar, los disparadores de este tipo reaccionan a los comandos DDL como: CREAR, MODIFICAR y SOLTAR
3. **Activadores de inicio de sesión:** el nombre lo dice todo. Este tipo reacciona a los eventos LOGON con usuarios.

Sintaxis de los disparadores.

La sintaxis SQL simplificada para definir el disparador es la siguiente.

```
CREATE TRIGGER [schema_name.]trigger_name
ON table_name
{FOR | AFTER | INSTEAD OF} {[INSERT] [,] [UPDATE] [,] [DELETE]}
AS
{sql_statements}
```

- Un conjunto de {sql_statements} que se realizará cuando se dispare el disparador (definido por los parámetros restantes).
- Debemos definir cuándo se dispara el disparador. Eso es lo que la parte {FOR | AFTER | INSTEAD OF}. Si nuestro disparador se define como FOR | AFTER | INSTEAD OF, las instrucciones SQL en el disparador se ejecutarán después de todas las acciones que disparó este disparador se lanzó con éxito. El INSTEAD OF realizará controles y reemplazará al acción original con la acción en el disparador, mientras que el FOR | AFTER (significan lo mismo) el disparador se ejecutará una vez completada la instrucción original.
- La parte {[INSERT] [,] [UPDATE] [,] [DELETE]} indica qué comando realmente activa este disparador. Debemos especificar al menos una opción, pero podríamos usar varias si la necesitamos.

B. Teoría preliminar sobre transacciones.

¿Qué son las transacciones?

Las transacciones agrupan un conjunto de tareas en una sola unidad de ejecución. Cada transacción comienza con una tarea específica y finaliza cuando todas las tareas del grupo se completan correctamente. Si alguna de las tareas falla, la transacción falla. Por lo tanto, **una transacción solo tiene dos resultados posibles, éxito o fracaso.**

Los pasos incompletos resultan en el fracaso de la transacción. Una transacción de base de datos, por definición, debe ser atómica, consistente, aislada y duradera.

¿Cómo implementar transacciones usando SQL?

Los siguientes comandos se utilizan para controlar las transacciones. Es importante tener en cuenta que estas instrucciones no se pueden usar al crear

tablas y solo se usan con los comandos de tipo DML como INSERT, UPDATE y DELETE.

1. **INICIAR TRANSACCIÓN:** Indica el punto de inicio de una transacción explícita o local.

```
START TRANSACTION transaction_name;
```

2. **ESTABLECER TRANSACCIÓN:** Coloca si la transacción será sólo de lectura o también se manejan escrituras.

```
SET TRANSACTION [ READ WRITE | READ ONLY ];
```

3. **COMPROMETER:** Si todo está en orden con todos los estados de cuenta dentro de una sola transacción, todos los cambios se registran juntos en la base de datos. El comando COMMIT guarda todas las transacciones en la base de datos desde el último comando COMMIT o ROLLBACK.

```
COMMIT;
```

Ejemplo de implementación del comando COMMIT.

Student				
Rol_No	Name	Address	Phone	Age
1	Ram	Delhi	9455123451	18
2	Ramesh	Gurgaon	9652431543	18
3	Sujit	Rohtak	9156253131	20
4	Suresh	Delhi	9156768971	18
3	Sujit	Rohtak	9156253131	20
2	Ramesh	Gurgaon	9652431543	18

Figura 1.1. Tabla denotada como 'Student' donde se almacena información relevante a los estudiantes de una institución.

A continuación se muestra un ejemplo donde se eliminaría los registros de la tabla que tienen edad igual a 20 años y luego se confirman los cambios en la base

de datos. Se realizaría mediante las siguientes instrucciones; primeramente iniciando una transacción.

```
START TRANSACTION transaction_name;
DELETE FROM Student WHERE AGE = 20;
COMMIT;
```

Por lo tanto, se eliminarían dos filas de la tabla y mediante una consulta con la instrucción SELECT, la tabla se vería de esta manera.

Rol_No	Name	Address	Phone	Age
1	Ram	Delhi	9455123451	18
2	Ramesh	Gurgaon	9652431543	18
4	Suresh	Delhi	9156768971	18
2	Ramesh	Gurgaon	9652431543	18

Figura 1.2. Tabla denotada como 'Student' donde se puede apreciar que los registros donde la edad de los estudiantes es igual a 20 años fueron eliminados satisfactoriamente, por lo tanto, los cambios dentro de la transacción fueron comprometidos.

- 4. ROLLBACK:** Si se produce algún error con cualquiera de las instrucciones agregadas de SQL, todos los cambios deben ser abortados. El proceso de revertir los cambios se denomina reversión. Este comando solo se puede usar para deshacer transacciones desde que se emitió el último comando COMMIT o ROLLBACK.

```
ROLLBACK;
```

Ejemplo de implementación del comando ROLLBACK.

Student				
Rol_No	Name	Address	Phone	Age
1	Ram	Delhi	9455123451	18
2	Ramesh	Gurgaon	9652431543	18
3	Sujit	Rohtak	9156253131	20
4	Suresh	Delhi	9156768971	18
3	Sujit	Rohtak	9156253131	20
2	Ramesh	Gurgaon	9652431543	18

Figura 1.4. Tabla denotada como 'Student' del ejemplo anterior.

Con respecto al ejemplo anterior de COMMIT, manejaremos la misma tabla de estudiantes, ahora vamos a eliminar los registros de la tabla donde los alumnos tengan 20 años y, posterior a eso, vamos a revertir los cambios en la base de datos.

```
START TRANSACTION transaction_name;  
DELETE FROM Student WHERE AGE = 20;  
ROLLBACK;
```

Como resultado, observaremos que las instrucciones que llevamos a cabo dentro de la transacción serán revertidas, por lo que los registros donde el alumno tenga 20 años, no serán eliminados.

Student				
Rol_No	Name	Address	Phone	Age
1	Ram	Delhi	9455123451	18
2	Ramesh	Gurgaon	9652431543	18
3	Sujit	Rohtak	9156253131	20
4	Suresh	Delhi	9156768971	18
3	Sujit	Rohtak	9156253131	20
2	Ramesh	Gurgaon	9652431543	18

Figura 1.5. Tabla denotada como 'Student' donde se puede apreciar que los registros permanecen igual como al principio de este ejemplo.

C. Práctica 4.1.

Competencia: El estudiante aprenderá a disparar una transacción y terminarla con commit o rollback.

Introducción: Al disparar una transacción se debe concluir con el grabado permanente de los cambios realizados, para ello se utiliza el commit, sin embargo, en caso de requerir deshacer los cambios puede aplicarse la sentencia rollback.

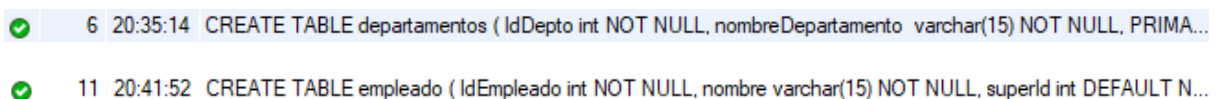
Metodología: Utilizando la base de datos 'mydb' realice las siguientes actividades:

Ejemplo de funcionamiento del COMMIT.

1. Primeramente, creamos las tablas dentro de la base de datos 'mydb' departamentos y empleado, mediante las siguientes instrucciones:

```
DROP TABLE IF EXISTS departamentos;  
CREATE TABLE departamentos  
(  
    IdDepto int NOT NULL,  
    nombreDepartamento varchar(15) NOT NULL,  
    PRIMARY KEY (IdDepto)  
) ENGINE = InnoDB;
```

```
CREATE TABLE empleado  
(  
    IdEmpleado int NOT NULL,  
    nombre varchar(15) NOT NULL,  
    superId int DEFAULT NULL,  
    salario double NOT NULL,  
    IdDepto int DEFAULT NULL,  
    PRIMARY KEY (IdEmpleado),  
  
    KEY deptosIdx (IdDepto),  
    CONSTRAINT deptosIdx  
    FOREIGN KEY (IdDepto)  
        REFERENCES departamentos (IdDepto)  
) ENGINE = InnoDB;
```



6	20:35:14	CREATE TABLE departamentos (IdDepto int NOT NULL, nombreDepartamento varchar(15) NOT NULL, PRIMA...
11	20:41:52	CREATE TABLE empleado (IdEmpleado int NOT NULL, nombre varchar(15) NOT NULL, superId int DEFAULT N...

Figura 2.1. Ejecución correcta de los comandos para crear las tablas 'departamentos' y 'empleado'.

2. Verifique el contenido de la tabla 'empleado' y la cantidad de registros mediante dos instrucciones SELECT.

```
SELECT * FROM empleado;  
SELECT count(*) FROM empleado;
```

	IdEmpleado	nombre	superId	salario	IdDepto	count(*)
*	NULL	NULL	NULL	NULL	NULL	0

```
✓ 13 20:44:00 SELECT * FROM empleado LIMIT 0, 1000  
✓ 14 20:44:23 SELECT count(*) FROM empleado LIMIT 0, 1000
```

Figura 2.2. Ejecución correcta de los comandos para verificar el contenido de la tabla 'empleado', como se acaba de crear, por lo tanto, se encuentra vacía.

3. Inicie una transacción (Cada una de las siguientes instrucciones deben ser realizadas de manera consecutiva hasta):

```
START TRANSACTION;
```

```
✓ 15 20:46:27 START TRANSACTION
```

Figura 2.3. Ejecución correcta del comando para iniciar una transacción.

4. Realice las siguientes inserciones:

```
INSERT INTO `mydb`.`empleado` (`IdEmpleado`, `nombre`, `salario`)  
VALUES ('100', 'CARLOS LOPEZ', '1000');  
INSERT INTO `mydb`.`empleado` (`IdEmpleado`, `nombre`, `salario`)  
VALUES ('200', 'JUAN PEREZ', '1000');
```

```
✓ 16 20:48:41 INSERT INTO `mydb`.`empleado` (`IdEmpleado`, `nombre`, `salario`) VALUES ('100', 'CARLOS LOPEZ', '1000')  
✓ 17 20:48:45 INSERT INTO `mydb`.`empleado` (`IdEmpleado`, `nombre`, `salario`) VALUES ('200', 'JUAN PEREZ', '1000')
```

Figura 2.4. Ejecución correcta de las inserciones realizadas a la tabla 'empleado'.

5. Verifique nuevamente la tabla 'empleado' para verificar si existen los datos:

```
SELECT * FROM empleado;
SELECT count(*) FROM empleado;
```

	IdEmpleado	nombre	superId	salario	IdDepto	
▶	100	CARLOS LOPEZ	NULL	1000	NULL	
	200	JUAN PEREZ	NULL	1000	NULL	count(*)
★	NULL	NULL	NULL	NULL	NULL	▶ 2

```
✓ 13 20:44:00 SELECT * FROM empleado LIMIT 0, 1000
✓ 14 20:44:23 SELECT count(*) FROM empleado LIMIT 0, 1000
```

Figura 2.5. Ejecución correcta de los comandos para verificar el contenido de la tabla 'empleado', como se observa, las dos inserciones que habíamos realizado anteriormente se encuentran dentro de la tabla, pero están entre el estado comprometido o revertido.

6. Comprometemos la información mediante la ejecución de COMMIT:

```
COMMIT;
```

```
✓ 19 20:52:44 SELECT count(*) FROM empleado LIMIT 0, 1000
```

Figura 2.6. Ejecución correcta del comando COMMIT, donde nos permitirá comprometer la información que ya se encuentra en la tabla.

7. Verifique nuevamente la tabla 'empleado' para verificar si existen los datos:

```
SELECT * FROM empleado;
SELECT count(*) FROM empleado;
```

	IdEmpleado	nombre	superId	salario	IdDepto	
▶	100	CARLOS LOPEZ	NULL	1000	NULL	
	200	JUAN PEREZ	NULL	1000	NULL	count(*)
★	NULL	NULL	NULL	NULL	NULL	▶ 2

```
✓ 13 20:44:00 SELECT * FROM empleado LIMIT 0, 1000
✓ 14 20:44:23 SELECT count(*) FROM empleado LIMIT 0, 1000
```

Figura 2.7. Ejecución correcta de los comandos para verificar el contenido de la tabla 'empleado', como se observa, se compromete la información y por lo tanto el estado de la transacción fue exitosa.

Explicación de los resultados obtenidos - Ejemplo de funcionamiento del COMMIT.

Con respecto a los resultados obtenidos, primeramente se creó una tabla dentro de la base de datos 'mydb' denotada como 'empleado' y se verificó la creación de la tabla mediante una consulta que cuenta los registros dentro de esta; se encuentra vacía.

Después de eso, se inicia la transacción, de manera consecutiva, se establecen algunas inserciones a la tabla 'empleado', después mediante una consulta verificamos si se encuentran dichas inserciones, las dos inserciones si aparecen dentro de la tabla, por lo que comprometemos la información dentro de la tabla realizando un COMMIT para grabar los cambios realizados de manera permanente; nuevamente verificamos y podemos observar que ya se establecieron de manera permanente.

Se pudo demostrar mediante la transacción que al ejecutar la instrucción COMMIT al final de realizar las instrucciones que se establecieron en la transacción fueron ejecutadas y los registros fueron asentados permanentemente, ya no podemos realizar un ROLLBACK para revertir esos registros.

Ejemplo de funcionamiento del ROLLBACK.

1. Inicie una transacción (Cada una de las siguientes instrucciones deben ser realizadas de manera consecutiva hasta):

START TRANSACTION;



Figura 3.1. Ejecución correcta del comando para iniciar una transacción.

2. Realizar las inserciones correspondientes:

```
INSERT INTO `mydb`.`empleado` (`IdEmpleado`, `nombre`, `salario`)
VALUES ('300', 'MARIA GOMEZ', '1000');
INSERT INTO `mydb`.`empleado` (`IdEmpleado`, `nombre`, `salario`)
VALUES ('400', 'GABRIELA ARENAS', '1000');
```

```
✓ 21 21:15:39 INSERT INTO `mydb`.`empleado` (`IdEmpleado`, `nombre`, `salario`) VALUES ('300', 'MARIA GOMEZ', '1000')
✓ 22 21:15:42 INSERT INTO `mydb`.`empleado` (`IdEmpleado`, `nombre`, `salario`) VALUES ('400', 'GABRIELA ARENAS', '1000')
```

Figura 3.2. Ejecución correcta de las inserciones realizadas a la tabla 'empleado'.

3. Verifique la tabla 'empleado' para verificar si existen los datos:

```
SELECT * FROM empleado;
SELECT count(*) FROM empleado;
```

	IdEmpleado	nombre	superId	salario	IdDepto	
▶	100	CARLOS LOPEZ	NULL	1000	NULL	
	200	JUAN PEREZ	NULL	1000	NULL	
	300	MARIA GOMEZ	NULL	1000	NULL	
	400	GABRIELA ARENAS	NULL	1000	NULL	
*	NULL	NULL	NULL	NULL	NULL	▶
						count(*)
						4

```
✓ 13 20:44:00 SELECT * FROM empleado LIMIT 0, 1000
✓ 14 20:44:23 SELECT count(*) FROM empleado LIMIT 0, 1000
```

Figura 3.3. Ejecución correcta de los comandos para verificar el contenido de la tabla 'empleado', como se observa, se encuentran los dos registros que anteriormente habíamos agregado, más los dos que agregamos en esta sección.

4. Hacemos una reversión de la información mediante un ROLLBACK:

```
ROLLBACK;
```

```
✓ 25 21:24:48 ROLLBACK
```

Figura 3.4. Ejecución correcta del ROLLBACK implementado.

5. Verifique nuevamente la tabla 'empleado' para verificar si existen los datos:

```
SELECT * FROM empleado;
SELECT count(*) FROM empleado;
```

	IdEmpleado	nombre	superId	salario	IdDepto	
▶	100	CARLOS LOPEZ	NULL	1000	NULL	
	200	JUAN PEREZ	NULL	1000	NULL	count(*)
•	NULL	NULL	NULL	NULL	NULL	▶ 2

- ✓ 13 20:44:00 SELECT * FROM empleado LIMIT 0, 1000
- ✓ 14 20:44:23 SELECT count(*) FROM empleado LIMIT 0, 1000

Figura 3.5. Ejecución correcta de los comandos para verificar el contenido de la tabla 'empleado', como se observa, las dos inserciones que habíamos realizado anteriormente fueron eliminadas por el ROLLBACK, por lo que solamente quedan las que hicimos en la anterior sección.

Explicación de los resultados obtenidos - Ejemplo de funcionamiento del ROLLBACK.

Con respecto a los resultados obtenidos, primero se empezó la transacción y se hicieron dos inserciones más a la tabla 'empleado', posteriormente, se ejecutó un ROLLBACK después de las inserciones, como se muestra en la figura 3.5, las inserciones que hicimos en esta sección fueron revertidas y por ende no permanecieron en la tabla; los datos no se asentaron permanentemente.

Se pudo demostrar mediante la transacción que al ejecutar la instrucción ROLLBACK al final de realizar las instrucciones que se establecieron en la transacción, la inserciones que fueron ejecutadas fueron abortadas y por ende la información comprometida para que posterior a la transacción sea agregada fue revertida.

Ejemplo de funcionamiento específico con ROLLBACK.

1. Inicie una nueva transacción (Cada una de las siguientes instrucciones deben ser realizadas de manera consecutiva hasta):

```
START TRANSACTION;
```

✓ 20 21:14:31 START TRANSACTION

Figura 4.1. Ejecución correcta del comando para iniciar una transacción.

2. Destruya la tabla 'empleado':

```
DROP TABLE empleado;
```

✓ 27 21:46:13 DROP TABLE empleado

Figura 4.2. Ejecución correcta del comando para destruir la tabla 'empleado'.

3. Realice un ROLLBACK:

```
ROLLBACK;
```

✓ 25 21:24:48 ROLLBACK

Figura 4.3. Ejecución correcta del ROLLBACK implementado.

4. Verifique nuevamente la tabla:

```
SELECT * FROM empleado;  
SELECT count(*) FROM empleado;
```

✗ 28 21:47:29 SELECT * FROM empleado LIMIT 0, 1000

Figura 4.4. Ejecución no satisfactoria de la consulta, debido a que la tabla 'empleado' no existe.

Explicación de los resultados obtenidos - Ejemplo de funcionamiento específico con ROLLBACK.

Con respecto a los resultados obtenidos, primero se empezó la transacción y después se eliminó la tabla 'empleado', posteriormente se realizó un ROLLBACK

para ver si era posible revertir esa instrucción, pero lamentablemente no fue posible debido a que el ROLLBACK solo atiende a las instrucciones de tipo DML (Lenguaje de manipulación de datos), pero como el DROP es una instrucción propia del DDL (Lenguaje de definición de datos) no era posible, por lo tanto la tabla fue eliminada.

Se pudo demostrar mediante la transacción que al ejecutar la instrucción ROLLBACK al final de realizar instrucciones de tipo DDL que se establecieron en la transacción, la reversión no se puede realizar; solamente funciona con instrucciones de tipo DML.

A. Ejemplo de la Práctica 4.1 con base de datos

‘maquina_expendedora’.

Ejemplo de funcionamiento del COMMIT.

1. Primeramente, verificamos el contenido de la tabla ‘cliente’ y la cantidad de registros que posee antes de realizar la transacción:

```
SELECT * FROM cliente;  
SELECT count(*) FROM cliente;
```

	id_cliente	id_institucion	nombre_cliente	apellido_paterno	apellido_materno		count(*)
▶	1	1010	Rodrigo	Barba	Navarro		
	20490687	1010	Sarah Collins	Romero	Spears		
	20490688	1010	Ashlee Durham	Nash	Patton		
	20490689	1010	Hashim Finley	Mccarty	Figueroa	▶	61

```
✓ 3 10:21:21 SELECT * FROM cliente LIMIT 0, 1000  
✓ 4 10:21:25 SELECT count(*) FROM cliente LIMIT 0, 1000
```

Figura 5.1. Ejecución correcta de los comandos para verificar el contenido de la tabla ‘cliente’, como se observa que tiene 61 registros en total, por lo que veremos cómo cambia con la transacción.

2. Posteriormente, iniciamos una transacción (Cada una de las siguientes instrucciones deben ser realizadas de manera consecutiva hasta):

```
START TRANSACTION;
```

7 10:26:23 START TRANSACTION

Figura 5.2. Ejecución satisfactoria de la transacción.

3. Realice las siguientes inserciones en la tabla 'cliente':

```
INSERT INTO `maquina expendedora`.`cliente` VALUES (2, 1010,
'Johan Antonio', 'Gurrola', 'Bernal');
INSERT INTO `maquina expendedora`.`cliente` VALUES (3, 1010,
'Alejandro', 'Lira', 'Lopez');
```

8 10:28:04 INSERT INTO `maquina expendedora`.`cliente` VALUES (2, 1010, 'Johan Antonio', 'Gurrola', 'Bernal')
9 10:28:06 INSERT INTO `maquina expendedora`.`cliente` VALUES (3, 1010, 'Alejandro', 'Lira', 'Lopez')

Figura 5.3. Ejecución correcta de las inserciones realizadas a la tabla 'cliente'.

4. Verificamos la tabla 'cliente' nuevamente para corroborar las inserciones:

```
SELECT * FROM cliente;
SELECT count(*) FROM cliente;
```

	id_cliente	id_institucion	nombre_cliente	apellido_paterno	apellido_materno		count(*)
▶	1	1010	Rodrigo	Barba	Navarro		
	2	1010	Johan Antonio	Gurrola	Bernal		
	3	1010	Alejandro	Lira	Lopez		
	20490687	1010	Sarah Collins	Romero	Spears	▶	63

10 10:28:51 SELECT * FROM cliente LIMIT 0, 1000
11 10:29:05 SELECT count(*) FROM cliente LIMIT 0, 1000

Figura 5.4. Ejecución correcta de los comandos para verificar el contenido de la tabla 'cliente', como se observa, ya son 63 registros en total; dos registros más en comparación a la otra consulta, siendo los dos registros nuevos los que acabamos de añadir.

5. Comprometemos la información mediante el comando COMMIT:

```
COMMIT;
```

✓ 19 20:52:44 SELECT count(*) FROM empleado LIMIT 0, 1000

Figura 5.5. Ejecución correcta del comando COMMIT, donde nos permitirá comprometer la información que ya se encuentra en la tabla 'cliente'.

6. Verificamos nuevamente la tabla 'cliente':

```
SELECT * FROM cliente;  
SELECT count(*) FROM cliente;
```

	id_cliente	id_institucion	nombre_cliente	apellido_paterno	apellido_materno		count(*)
▶	1	1010	Rodrigo	Barba	Navarro		
	2	1010	Johan Antonio	Gurrola	Bernal		
	3	1010	Alejandro	Lira	Lopez		
	20490687	1010	Sarah Collins	Romero	Spears	▶	63

✓ 10 10:28:51 SELECT * FROM cliente LIMIT 0, 1000

✓ 11 10:29:05 SELECT count(*) FROM cliente LIMIT 0, 1000

Figura 5.6. Ejecución correcta de los comandos para verificar el contenido de la tabla 'cliente', como se observa, mediante el COMMIT, pudimos grabar las inserciones de manera permanente.

Explicación de los resultados obtenidos - Ejemplo de funcionamiento del COMMIT.

Con respecto a los resultados obtenidos, se pudo observar que al iniciar la transacción y posteriormente realizar las inserciones, a la tabla 'cliente' fueron agregados dos registros más, pero la información todavía no estaba comprometida, por lo que cualquier error dentro de la transacción podría provocar que se deshicieran dichos registros; mediante el comando COMMIT, logramos grabar esos registros de manera permanente, como se observa en la figura 5.6.

Ejemplo de funcionamiento del ROLLBACK.

1. Primeramente, iniciamos una transacción (Cada una de las siguientes instrucciones deben ser realizadas de manera consecutiva hasta):

START TRANSACTION;

7 10:26:23 START TRANSACTION

Figura 6.1. Ejecución satisfactoria de la transacción.

2. Posteriormente, realizamos las siguientes inserciones en la tabla 'cliente':

```
INSERT INTO `maquina expendedora`.`cliente` VALUES (4, 1010,
'Luis', 'Barba', 'Navarro');
INSERT INTO `maquina expendedora`.`cliente` VALUES (5, 1010,
'Mario', 'Navarro', 'Ruíz');
```

13 10:52:52 INSERT INTO `maquina expendedora`.`cliente` VALUES (4, 1010, 'Luis', 'Barba', 'Navarro')
14 10:52:55 INSERT INTO `maquina expendedora`.`cliente` VALUES (5, 1010, 'Mario', 'Navarro', 'Ruíz')

Figura 6.2. Ejecución correcta de las inserciones realizadas a la tabla 'cliente'.

3. Verificamos la tabla 'cliente':

```
SELECT * FROM cliente;
SELECT count(*) FROM cliente;
```

	id_cliente	id_institucion	nombre_cliente	apellido_paterno	apellido_materno		count(*)
▶	1	1010	Rodrigo	Barba	Navarro		
	2	1010	Johan Antonio	Gurrola	Bernal		
	3	1010	Alejandro	Lira	Lopez		
	4	1010	Luis	Barba	Navarro		
	5	1010	Mario	Navarro	Ruíz	▶	65

15 10:54:00 SELECT * FROM cliente LIMIT 0, 1000
16 10:54:24 SELECT count(*) FROM cliente LIMIT 0, 1000

Figura 6.3. Ejecución correcta de los comandos para verificar el contenido de la tabla 'cliente', como se observa, ahora son 65 registros en total en la tabla 'cliente', lo que anteriormente eran 63 registros, sin embargo, la información no se encuentra comprometida.

4. Realizamos una reversión de los cambios con ROLLBACK:

`ROLLBACK;`

✓ 25 21:24:48 ROLLBACK

Figura 6.4. Ejecución correcta del ROLLBACK implementado.

5. Verificamos nuevamente la tabla 'cliente':

```
SELECT * FROM cliente;  
SELECT count(*) FROM cliente;
```

	id_cliente	id_institucion	nombre_cliente	apellido_paterno	apellido_materno		count(*)
▶	1	1010	Rodrigo	Barba	Navarro		
	2	1010	Johan Antonio	Gurrola	Bernal		
	3	1010	Alejandro	Lira	Lopez		
	20490687	1010	Sarah Collins	Romero	Spears	▶	63

✓ 15 10:54:00 SELECT * FROM cliente LIMIT 0, 1000

✓ 16 10:54:24 SELECT count(*) FROM cliente LIMIT 0, 1000

Figura 6.5. Ejecución correcta de los comandos para verificar el contenido de la tabla 'cliente', como se observa, los registros que anteriormente agregamos fueron desechados, por ende, las inserciones realizadas dentro de la transacción fueron revertidas.

Explicación de los resultados obtenidos - Ejemplo de funcionamiento del ROLLBACK.

Con respecto a los resultados obtenidos, se pudo observar que al iniciar la transacción y luego realizar las inserciones, a la tabla 'cliente' fueron agregados dos registros más, pero la información todavía no estaba comprometida; mediante el comando ROLLBACK, logramos descartar las inserciones que habíamos hecho anteriormente, como se observa en la figura 6.5, por lo que resulta útil implementar un comando ROLLBACK en caso de que se desee revertir los cambios hecho en una tabla.

Ejemplo de funcionamiento específico con ROLLBACK.

1. Primeramente, iniciamos una transacción (Cada una de las siguientes instrucciones deben ser realizadas de manera consecutiva hasta):

```
START TRANSACTION;
```

7 10:26:23 START TRANSACTION

Figura 7.1. Ejecución satisfactoria de la transacción.

2. Posteriormente, verificamos que la tabla 'tarjeta' realmente existe:

```
SELECT * FROM tarjeta;
```

```
SELECT count(*) FROM tarjeta;
```

	id_tarjeta	id_cliente	fecha_expedicion	puntos	estatus		
▶	10020512	20490687	2022-05-10	170	Habilitada		
	10020513	20490688	2022-05-14	50	Habilitada		
	10020514	20490689	2022-05-18	325	Habilitada		
	10020515	20490690	2022-05-22	707	Habilitada	▶	count(*)
							150

31 11:36:15 SELECT * FROM tarjeta LIMIT 0, 1000

32 11:36:30 SELECT count(*) FROM tarjeta LIMIT 0, 1000

Figura 7.2. Ejecución correcta de los comandos para verificar el contenido de la tabla 'tarjeta'.

3. Destruimos la tabla 'tarjeta':

```
DROP TABLE tarjeta;
```

34 11:37:12 DROP TABLE tarjeta

Figura 7.3. Ejecución correcta de los comandos para eliminar la tabla 'tarjeta'.

4. Realizamos una reversión con el comando ROLLBACK:

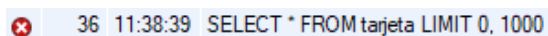
```
ROLLBACK;
```

25 21:24:48 ROLLBACK

Figura 7.4. Ejecución correcta del ROLLBACK implementado.

5. Verificamos si la tabla 'institucion' aún existe.

```
SELECT * FROM institucion;  
SELECT count(*) FROM institucion;
```



36 11:38:39 SELECT * FROM tarjeta LIMIT 0, 1000

Figura 7.5. Ejecución no satisfactoria de la consulta, debido a que la tabla 'tarjeta' no existe.

Explicación de los resultados obtenidos - Ejemplo de funcionamiento específico con ROLLBACK.

Con respecto a los resultados obtenidos, primero se empezó la transacción y después se eliminó la tabla 'tarjeta', posteriormente se realizó un ROLLBACK para ver si era posible revertir esa instrucción, pero lamentablemente no fue posible debido a que el ROLLBACK solo atiende a las instrucciones de tipo DML, por lo tanto la tabla fue eliminada.

Se pudo demostrar mediante la transacción que al ejecutar la instrucción ROLLBACK al final de realizar instrucciones de tipo DDL que se establecieron en la transacción, la reversión no se puede realizar; solamente funciona con instrucciones de tipo DML.

B. Práctica 4.2.

Competencia: El estudiante aprenderá a disparar una transacción y terminarla con commit o rollback.

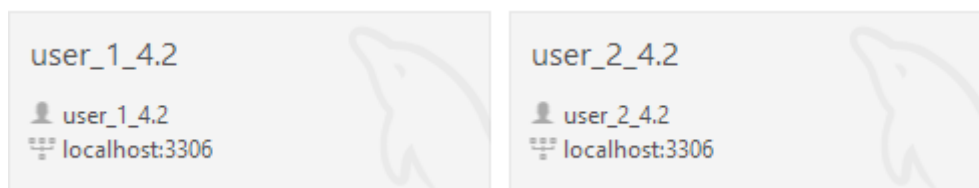
Introducción: Al disparar una transacción se debe concluir con el grabado permanente de los cambios realizados, para ello se utiliza el commit, sin embargo, en caso de requerir deshacer los cambios puede aplicarse la sentencia rollback.

Metodología: Utilizando la base de datos mydb y la tabla empleado realice las siguientes actividades:

Primeramente, creamos dos usuario con el nombre de usuario user_1_4.2 y user_2_4.2, ambos se conectan mediante el host local, y su contraseña es la clásica numeración del 1 al 5. Poseen los privilegios de todo el lenguaje de manipulación y definición de datos sobre la base de datos 'maquina_expendedora' y 'mydb', como se muestra en la siguiente imagen.

Schema	Privileges
maquina_expendedora	ALTER, ALTER ROUTINE, CREATE, CREATE ROUTINE, CREATE VIEW, DELETE, EXECUTE, INDEX, INSERT,
mydb	ALTER, ALTER ROUTINE, CREATE, CREATE ROUTINE, CREATE VIEW, DELETE, EXECUTE, INDEX, INSERT,

Posteriormente, se establecieron conexiones con dichos usuarios, como se muestra en la siguiente imagen, al iniciar sesión nos pedirá contraseña, sólo la indicamos y damos en el botón continuar.



Ejemplo de funcionamiento de inserción con diferentes usuarios.

1. Primeramente, iniciamos sesión desde la conexión de user_1_4.2, y hacemos lo siguiente.
2. Iniciamos una transacción:

`START TRANSACTION;`

7 10:26:23 START TRANSACTION

Figura 8.1. Ejecución satisfactoria de la transacción.

3. Realizamos las siguientes inserciones:

```
INSERT INTO `mydb`.`empleado` (`IdEmpleado`, `nombre`, `salario`)
VALUES ('100', 'CARLOS LOPEZ', '1000');
INSERT INTO `mydb`.`empleado` (`IdEmpleado`, `nombre`, `salario`)
VALUES ('200', 'JUAN PEREZ', '1000');
```

```
✓ 10 13:32:59 INSERT INTO `mydb`.`empleado` (`IdEmpleado`, `nombre`, `salario`) VALUES ('100', 'CARLOS LOPEZ', '1000')
✓ 11 13:33:01 INSERT INTO `mydb`.`empleado` (`IdEmpleado`, `nombre`, `salario`) VALUES ('200', 'JUAN PEREZ', '1000')
```

Figura 8.2. Ejecución satisfactoria de algunas inserciones a la tabla 'empleado', desde el usuario user_1_4.2.

4. Verificamos el contenido de la tabla y la cantidad de registros:

```
SELECT * FROM empleado;
SELECT count(*) FROM empleado;
```

	IdEmpleado	nombre	superId	salario	IdDepto		count(*)
▶	100	CARLOS LOPEZ	NULL	1000	NULL		
	200	JUAN PEREZ	NULL	1000	NULL	▶	2

```
✓ 12 13:33:39 SELECT * FROM empleado LIMIT 0, 1000
✓ 13 13:33:57 SELECT count(*) FROM empleado LIMIT 0, 1000
```

Figura 8.3. Ejecución correcta de los comandos para verificar el contenido de la tabla 'empleado', donde podemos observar que los registros se encuentran en la tabla, pero no están comprometidos.

5. Ahora iniciamos sesión con user_2_4.2.

6. Iniciamos una nueva transacción:

```
START TRANSACTION;
```

```
✓ 7 10:26:23 START TRANSACTION
```

Figura 8.4. Ejecución satisfactoria de la transacción.

7. Verificamos el contenido de la tabla y la cantidad de registros:

```
SELECT * FROM empleado;
SELECT count(*) FROM empleado;
```

	IdEmpleado	nombre	superId	salario	IdDepto	count(*)
*	NULL	NULL	NULL	NULL	NULL	0

✓	3	13:35:00	SELECT * FROM empleado LIMIT 0, 1000
✓	4	13:35:13	SELECT count(*) FROM empleado LIMIT 0, 1000

Figura 8.5. Ejecución correcta de los comandos para verificar el contenido de la tabla 'empleado', donde podemos observar que se encuentra vacía, debido a que las inserciones que hicimos desde el usuario user_1_4.2, no fueron comprometidas.

8. Realizamos las siguientes inserciones:

```
INSERT INTO `mydb`.`empleado` (`IdEmpleado`, `nombre`, `salario`)
VALUES ('300', 'MARIA GOMEZ', '1000');
INSERT INTO `mydb`.`empleado` (`IdEmpleado`, `nombre`, `salario`)
VALUES ('400', 'GABRIELA ARENAS', '1000');
```

✓	5	13:35:47	INSERT INTO `mydb`.`empleado` (`IdEmpleado`, `nombre`, `salario`) VALUES ('300', 'MARIA GOMEZ', '1000')
✓	6	13:35:50	INSERT INTO `mydb`.`empleado` (`IdEmpleado`, `nombre`, `salario`) VALUES ('400', 'GABRIELA ARENAS', '1000')

Figura 8.6. Ejecución satisfactoria de algunas inserciones a la tabla 'empleado' desde el usuario user_2_4.2.

9. Ahora regresamos a user_1_4.2

10. Verificamos el contenido de la tabla y la cantidad de registros:

```
SELECT * FROM empleado;
SELECT count(*) FROM empleado;
```

	IdEmpleado	nombre	superId	salario	IdDepto	
▶	100	CARLOS LOPEZ	NULL	1000	NULL	
	200	JUAN PEREZ	NULL	1000	NULL	
*	NULL	NULL	NULL	NULL	NULL	▶
						count(*)
						2

- ✓ 14 13:36:15 SELECT * FROM empleado LIMIT 0, 1000
- ✓ 15 13:36:34 SELECT count(*) FROM empleado LIMIT 0, 1000

Figura 8.7. Ejecución correcta de los comandos para verificar el contenido de la tabla 'empleado', donde podemos observar que los datos que habíamos insertado antes, siguen ahí sin comprometerse.

11. Comprometemos los datos con comando COMMIT:

COMMIT;

- ✓ 16 13:36:57 COMMIT

Figura 8.8. Ejecución correcta del comando COMMIT, donde nos permitirá comprometer la información que ya se encuentra en la tabla desde el usuario user_1_4.2.

12. Verificamos el contenido de la tabla y la cantidad de registros:

SELECT * FROM empleado;
SELECT count(*) FROM empleado;

	IdEmpleado	nombre	superId	salario	IdDepto	
▶	100	CARLOS LOPEZ	NULL	1000	NULL	
	200	JUAN PEREZ	NULL	1000	NULL	
*	NULL	NULL	NULL	NULL	NULL	▶
						count(*)
						2

- ✓ 17 13:37:15 SELECT * FROM empleado LIMIT 0, 1000
- ✓ 18 13:38:14 SELECT count(*) FROM empleado LIMIT 0, 1000

Figura 8.9. Ejecución correcta de los comandos para verificar el contenido de la tabla 'empleado', donde podemos observar que los datos ya se encuentran comprometidos.

13. Ahora regresamos a user_2_4.2

14. Verificamos el contenido de la tabla y la cantidad de registros:

```
SELECT * FROM empleado;
SELECT count(*) FROM empleado;
```

	IdEmpleado	nombre	superId	salario	IdDepto	
▶	300	MARIA GOMEZ	NULL	1000	NULL	
	400	GABRIELA ARENAS	NULL	1000	NULL	
*	NULL	NULL	NULL	NULL	NULL	count(*)
						2

- ✓ 17 13:37:15 SELECT * FROM empleado LIMIT 0, 1000
- ✓ 18 13:38:14 SELECT count(*) FROM empleado LIMIT 0, 1000

Figura 8.10. Ejecución correcta de los comandos para verificar el contenido de la tabla 'empleado', donde podemos observar que los datos que habíamos insertado antes, siguen ahí sin comprometerse.

15. Comprometemos los datos con comando COMMIT:

```
COMMIT;
```

- ✓ 16 13:36:57 COMMIT

Figura 8.11. Ejecución correcta del comando COMMIT, donde nos permitirá comprometer la información que ya se encuentra en la tabla desde el usuario user_2_4.2.

16. Verificamos el contenido de la tabla y la cantidad de registros:

```
SELECT * FROM empleado;
SELECT count(*) FROM empleado;
```

	IdEmpleado	nombre	superId	salario	IdDepto	
▶	100	CARLOS LOPEZ	NULL	1000	NULL	
	200	JUAN PEREZ	NULL	1000	NULL	
	300	MARIA GOMEZ	NULL	1000	NULL	
	400	GABRIELA ARENAS	NULL	1000	NULL	
*	NULL	NULL	NULL	NULL	NULL	count(*)
						4

- ✓ 17 13:37:15 SELECT * FROM empleado LIMIT 0, 1000
- ✓ 18 13:38:14 SELECT count(*) FROM empleado LIMIT 0, 1000

Figura 8.12. Ejecución correcta de los comandos para verificar el contenido de la tabla 'empleado', donde podemos observar que los datos ya se encuentran comprometidos, tanto los de las inserciones realizadas por el usuario user_1_4.2, como el usuario user_2_4.2.

17. Ahora regresamos a user_1_4.2

18. Verificamos el contenido de la tabla y la cantidad de registros:

```
SELECT * FROM empleado;
SELECT count(*) FROM empleado;
```

	IdEmpleado	nombre	superId	salario	IdDepto	
▶	100	CARLOS LOPEZ	NULL	1000	NULL	
	200	JUAN PEREZ	NULL	1000	NULL	
	300	MARIA GOMEZ	NULL	1000	NULL	
	400	GABRIELA ARENAS	NULL	1000	NULL	
*	NULL	NULL	NULL	NULL	NULL	

	count(*)
▶	4

✓	17	13:37:15	SELECT * FROM empleado LIMIT 0, 1000
✓	18	13:38:14	SELECT count(*) FROM empleado LIMIT 0, 1000

Figura 8.13. Ejecución correcta de los comandos para verificar el contenido de la tabla 'empleado', donde podemos observar que los datos ya todos los datos se encuentran comprometidos.

Explicación de los resultados obtenidos - Funcionamiento de inserción con diferentes usuarios.

Con base a los resultados obtenidos, se puede decir que las transacciones realizadas por dos diferentes usuarios trabajaron de manera aislada, eso quiere decir que las modificaciones de recursos o datos realizadas por diferentes transacciones son separadas. Con el usuario user_1_4.2 se hicieron dos inserciones a la tabla 'empleado', solamente se podían ver los cambios realizados desde ese usuario pero no estaba comprometida de manera general la información, por otra parte, con el usuario user_2_4.2, no se podía ver dichos cambios porque no habíamos hecho COMMIT desde el usuario user_1_4.2 todavía.

Asimismo, desde el usuario user_2_4.2 se hicieron dos inserciones pero tampoco se comprometieron los datos con COMMIT para verificar desde el usuario

user_1_4.2 si dichas inserciones aparecían, pero no resultó debido a que la información seguía sin comprometerse.

Posteriormente, realizamos primero el comprometimiento de la información con un COMMIT desde el usuario user_1_4.2; en sí, desde el mismo usuario no había tanta diferencia, pero en el usuario user_2_4.2 tampoco había diferencia, pero ¿a qué se debía? si ya habíamos realizado un COMMIT desde el primer usuario. Básicamente, estábamos bajo la presencia de una lectura consistente, eso quiere decir que, como el usuario user_2_4.2 seguía con una transacción en progreso, los cambios realizados desde el usuario user_1_4.2 no iban a tomar efecto hasta que finaliza la transacción reciente.


Es por tal, cuando realizamos el comprometimiento de la información con un COMMIT desde el usuario user_2_4.2, al realizar una consulta SELECT, podemos observar que ya las inserciones del user_1_4.2 ya aparecen, tanto para los dos usuarios. Por lo que se puede decir que, es importante saber cuándo finalizar una transacción.

Ejemplo de funcionamiento de actualización de datos con diferentes usuarios.

Vamos a establecer conexiones con los usuarios user_1_4.2 y user_1_4.2 creados anteriormente, y serán utilizados durante toda esta sección para realizar este ejemplo de funcionamiento; siendo así más eficientes.

1. Primeramente, vamos a iniciar sesión con user_1_4.2.
2. Iniciamos una transacción:

```
START TRANSACTION;
```



2 17:32:51 START TRANSACTION

Figura 9.1. Ejecución satisfactoria de la transacción.

3. Realizamos la siguiente actualización de los datos:

```
UPDATE empleado SET salario=6000 WHERE IdEmpleado=400;
```

✓ 3 17:33:18 UPDATE empleado SET salario=6000 WHERE IdEmpleado=400

Figura 9.2. Ejecución satisfactoria de la actualización del empleado con identificador 400 en la tabla 'empleado', desde el usuario user_1_4.2.

4. Verifique el contenido de la tabla y la cantidad de registros:

```
SELECT * FROM empleado;  
SELECT count(*) FROM empleado;
```

	IdEmpleado	nombre	superId	salario	IdDepto
▶	100	CARLOS LOPEZ	NULL	1000	NULL
	200	JUAN PEREZ	NULL	1000	NULL
	300	MARIA GOMEZ	NULL	1000	NULL
	400	GABRIELA ARENAS	NULL	6000	NULL
⌵	NULL	NULL	NULL	NULL	NULL

✓ 4 17:33:52 SELECT * FROM empleado LIMIT 0, 1000

Figura 9.3. Ejecución correcta de los comandos para verificar el contenido de la tabla 'empleado', donde podemos observar que aparecen los cambios propios que se hicieron más no se han comprometido dichos datos.

5. Después, iniciamos sesión con user_2_4.2.

6. Posterior a eso, iniciamos una nueva transacción:

```
START TRANSACTION;
```

✓ 1 17:37:06 START TRANSACTION

Figura 9.4. Ejecución satisfactoria de la transacción.

7. Verifique el contenido de la tabla y la cantidad de registros:

```
SELECT * FROM empleado;
SELECT count(*) FROM empleado;
```

	IdEmpleado	nombre	superId	salario	IdDepto
▶	100	CARLOS LOPEZ	NULL	1000	NULL
	200	JUAN PEREZ	NULL	1000	NULL
	300	MARIA GOMEZ	NULL	1000	NULL
	400	GABRIELA ARENAS	NULL	1000	NULL

✓ 4 17:37:41 SELECT * FROM empleado LIMIT 0, 1000

Figura 9.5. Ejecución correcta de los comandos para verificar el contenido de la tabla 'empleado', donde podemos observar que los cambios realizados al salario no aparecen.

8. Realizamos la siguiente actualización de los datos:

```
UPDATE empleado SET salario=4500 WHERE IdEmpleado=400;
```

4 5 17:38:16 UPDATE empleado SET salario=4500 WHERE IdEmpleado=400

Running...

Figura 9.6. Ejecución no satisfactoria de la actualización del empleado con identificador 400 en la tabla 'empleado', desde el usuario user_2_4.2.

¿Qué está ocurriendo? Debido a que tenemos una transacción en progreso que está manejando el mismo registro desde el usuario user_1_4.2, no nos permite actualizar dicho registro hasta que dicha transacción concluya. Mientras tanto, en el registro de ejecución aparece la leyenda 'Running' pero no llega a completarse.

9. Después, regresamos con user_1_4.2.

10. Comprometemos los datos con comando COMMIT:

```
COMMIT;
```

✓ 6 17:39:12 COMMIT

Figura 9.7. Ejecución correcta del comando COMMIT, donde nos permitirá comprometer la información que ya se encuentra actualizada en la tabla desde el usuario user_1_4.2.

11. Después, regresamos con user_2_4.2.

¿Qué sucedió con la actualización? En sí, debido a que pasó un tiempo determinado en 'Running', automáticamente paró y mostró el error de que no fue posible completar la instrucción en ese momento. Después de eso, si hacemos la actualización, si nos permite de manera instantánea, debido a que ya concluyó la transacción anterior del usuario user_1_4.2

12. Comprometemos los datos con comando COMMIT:

COMMIT;

```
✓ 6 17:40:09 UPDATE empleado SET salario=4500 WHERE IdEmpleado=400
✓ 7 17:40:16 COMMIT
```

Figura 9.8. Ejecución correcta del comando COMMIT, donde nos permitirá comprometer la información que ya se encuentra actualizada en la tabla desde el usuario user_2_4.2.

13. Verifique el contenido de la tabla y la cantidad de registros:

```
SELECT * FROM empleado;
SELECT count(*) FROM empleado;
```

	IdEmpleado	nombre	superId	salario	IdDepto
▶	100	CARLOS LOPEZ	NULL	1000	NULL
	200	JUAN PEREZ	NULL	1000	NULL
	300	MARIA GOMEZ	NULL	1000	NULL
	400	GABRIELA ARENAS	NULL	4500	NULL

```
✓ 8 17:40:47 SELECT * FROM empleado LIMIT 0, 1000
```

Figura 9.9. Ejecución correcta de los comandos para verificar el contenido de la tabla 'empleado', donde podemos observar que conservó la actualización de la última transacción concluida, o sea, la del usuario user_2_4.2.

Explicación de los resultados obtenidos - Funcionamiento de actualización de datos con diferentes usuarios.

Con base a los resultados obtenidos, como sucedió con la sección anterior a esta, se puede decir que las transacciones realizadas por dos diferentes usuarios trabajaron de manera aislada. Con el usuario user_1_4.2 se hizo una actualización; solamente se podían ver los cambios realizados desde ese usuario pero no estaba comprometida de manera general la información, por otra parte, con el usuario user_2_4.2, no se podía ver dichos cambios porque no habíamos hecho COMMIT desde dicho usuario todavía.

Asimismo, desde el usuario user_2_4.2 se intentó realizar una actualización del mismo registro afectado desde la transacción del usuario user_1_4.2, pero no nos permitió, de hecho, la instrucción no finalizó y se quedó congelado con la leyenda 'Running', esto debido a que ya se encontraba una transacción en progreso que consecuentemente manejaba ese registro.

Al final, simplemente se cambió a la sesión de usuario user_1_4.2 para concluir la transacción mediante el comando COMMIT, y ahora, desde la sesión del usuario user_2_4.2, ya se veía la actualización hecha por la transacción reciente. Desde ese momento, ya se nos permitía realizar la actualización desde el usuario user_2_4.2, por lo que ejecutamos el comando COMMIT, y permaneció la última actualización de la transacción más reciente.

C. Ejemplo de la Práctica 4.2 con base de datos 'maquina_expendedora'.

1. Primeramente, iniciamos sesión con user_1_4.2.
2. Después, iniciamos una transacción:

```
START TRANSACTION;
```

✓ 2 17:32:51 START TRANSACTION

Figura 10.1. Ejecución satisfactoria de la transacción.

3. Realizamos la siguiente actualización de los datos:

```
UPDATE tarjeta SET puntos=7777 WHERE id_cliente=20490687 AND
estatus = "Habilitada";
```

✓ 6 19:26:41 UPDATE tarjeta SET puntos=7777 WHERE id_cliente=20490687 AND estatus = "Habilitada"

Figura 10.2. Ejecución satisfactoria de la actualización de la tarjeta del cliente con identificador 20490687 y que se encuentre habilitada dicha tarjeta, en la tabla 'tarjeta', desde el usuario user_1_4.2.

4. Verificamos el contenido de la tabla y la cantidad de registros:

```
SELECT * FROM tarjeta;
SELECT count(*) FROM tarjeta;
```

	id_tarjeta	id_cliente	fecha_expedicion	puntos	estatus
▶	10020512	20490687	2022-05-10	7777	Habilitada
	10020513	20490688	2022-05-14	50	Habilitada
	10020514	20490689	2022-05-18	325	Habilitada

✓ 7 19:27:23 SELECT * FROM tarjeta LIMIT 0, 1000

Figura 10.3. Ejecución correcta de los comandos para verificar el contenido de la tabla 'tarjeta', donde podemos observar que aparecen los cambios propios que se hicieron más no se han comprometido dichos datos.

5. Después, iniciamos sesión con user_2_4.2

6. Iniciamos una nueva transacción:

```
START TRANSACTION;
```

2 17:32:51 START TRANSACTION

Figura 10.4. Ejecución satisfactoria de la transacción.

7. Verificamos el contenido de la tabla y la cantidad de registros:

```
SELECT * FROM tarjeta;  
SELECT count(*) FROM tarjeta;
```

	id_tarjeta	id_cliente	fecha_expedicion	puntos	estatus
▶	10020512	20490687	2022-05-10	293	Habilitada
	10020513	20490688	2022-05-14	50	Habilitada
	10020514	20490689	2022-05-18	325	Habilitada

5 19:44:00 SELECT * FROM tarjeta LIMIT 0, 1000

Figura 10.5. Ejecución correcta de los comandos para verificar el contenido de la tabla 'cliente', donde podemos observar que los cambios realizados a los puntos no aparecen.

8. Realizamos la siguiente actualización de los datos:

```
UPDATE tarjeta SET puntos=1 WHERE id_cliente=20490687 AND estatus  
= "Habilitada";
```

4 6 19:44:30 UPDATE tarjeta SET puntos=1 WHERE id_cliente=20490687 AND estatus = "Habilitada" Running...

Figura 10.6. Ejecución no satisfactoria de la actualización de la tarjeta de cliente con identificador 20490687 y que esté habilitada, en la tabla 'cliente', desde el usuario user_2_4.2.

¿Qué está ocurriendo? Ya que tenemos una transacción en progreso que está manejando el mismo registro desde el usuario user_1_4.2, no nos permite actualizar dicho registro hasta que dicha transacción concluya.

9. Posterior a eso, regresamos a la sesión con user_1_4.2.

10. Comprometemos los datos con comando COMMIT:

```
COMMIT;
```


✓ 5 19:44:58 COMMIT

Figura 10.7. Ejecución correcta del comando COMMIT, donde nos permitirá comprometer la información que ya se encuentra actualizada en la tabla desde el usuario user_1_4.2.

11. Regresamos a la sesión con user_2_4.2.

¿Qué sucedió con la actualización? Debido a que pasó un tiempo determinado en 'Running', automáticamente paró y mostró el error de que no fue posible completar la instrucción en ese momento. Después de eso, si hacemos la actualización, si nos permite de manera instantánea, debido a que ya concluyó la transacción anterior del usuario user_1_4.2

12. Comprometemos los datos con comando COMMIT:

COMMIT;

✓ 6 19:44:30 UPDATE tarjeta SET puntos=1 WHERE id_cliente=20490687 AND estatus = "Habilitada"
✓ 7 19:45:34 COMMIT

Figura 10.8. Ejecución correcta del comando COMMIT, donde nos permitirá comprometer la información que ya se encuentra actualizada en la tabla desde el usuario user_2_4.2.

13. Verificamos el contenido de la tabla y la cantidad de registros:

SELECT * FROM tarjeta;

SELECT count(*) FROM tarjeta;

	id_tarjeta	id_cliente	fecha_expedicion	puntos	estatus
▶	10020512	20490687	2022-05-10	1	Habilitada
	10020513	20490688	2022-05-14	50	Habilitada
	10020514	20490689	2022-05-18	325	Habilitada

✓ 8 19:45:48 SELECT * FROM tarjeta LIMIT 0, 1000

Figura 10.9. Ejecución correcta de los comandos para verificar el contenido de la tabla 'cliente', donde podemos observar que conservó la actualización de la última transacción concluida, o sea, la del usuario user_2_4.2.

Explicación de los resultados obtenidos - Ejemplo de base de datos 'maquina expendedora'.

Como sucedió con la sección anterior a esta, se puede decir que las transacciones realizadas por dos diferentes usuarios trabajaron de manera aislada.

El usuario user_2_4.2 se intentó realizar una actualización del mismo registro afectado desde la transacción del usuario user_1_4.2, pero no nos permitió, esto debido a que ya se encontraba una transacción en progreso que consecuentemente manejaba ese registro.

Al final, simplemente se cambió a la sesión de usuario user_1_4.2 para concluir la transacción mediante el comando COMMIT, y ahora, desde la sesión del usuario user_2_4.2, ya se veía la actualización hecha por la transacción reciente.

Desde ese momento, ya se nos permitía realizar la actualización desde el usuario user_2_4.2, por lo que ejecutamos el comando COMMIT, y permaneció la última actualización de la transacción más reciente.

D. Práctica 5.1 sobre procedimientos almacenados.

Competencia:

Crear, destruir y ejecutar un Procedimiento almacenado.

Introducción:

Un procedimiento almacenado (SP) es un programa que se guarda físicamente en una base de datos, para crearlos se utiliza la instrucción `CREATE PROCEDURE nombre_proc`, para destruirlo la instrucción `DROP PROCEDURE nombre_proc`, para invocarlo la instrucción `CALL nombre_proc`.

Un SP permite el uso de parámetros de tres tipos, IN, OUT e INOUT, los parámetros IN son para enviar información al SP, mientras que los parámetros OUT permiten enviar información a las variables usadas en estos parámetros.

Procedimiento												
<pre>DROP PROCEDURE IF EXISTS ReverseProcedure; DELIMITER // CREATE PROCEDURE ReverseProcedure (IN String VARCHAR(80), OUT Reversed VARCHAR(80), OUT Length INTEGER) BEGIN SET Reversed = REVERSE(String); SET Length = LENGTH(String); SELECT String, Reversed, Length; END // DELIMITER ;</pre>												
Demostración												
<pre>CALL ReverseProcedure('Herong', @Reversed, @Length); SELECT @n:=nombre from clientes where noCliente=10015; CALL ReverseProcedure(@n, @Reversed, @Length);</pre>												
Resultados												
<table><tr><th></th><th>String</th><th>Reversed</th><th>Length</th><th></th><th>@n:=nombre</th></tr><tr><td>▶</td><td>Herong</td><td>gnoreH</td><td>6</td><td>▶</td><td>RACSO</td></tr></table>		String	Reversed	Length		@n:=nombre	▶	Herong	gnoreH	6	▶	RACSO
	String	Reversed	Length		@n:=nombre							
▶	Herong	gnoreH	6	▶	RACSO							
Explicación de resultados												
<p>Se creó un procedimiento llamado Reverse Procedure el cual cuenta con la variable string con parámetros de entrada, también cuenta con las variables Reversed, Length con parámetros de salida, comienza el procedimiento. La cláusula SET especifica las columnas que se deben actualizar y los valores nuevos para las columnas por lo cual almacenará el nuevo valor de Reversed después de cumplir con la función reverse (string) lo cual nos muestra el nombre invertido de la variable String. La cláusula SET especifica las columnas que se deben actualizar y los valores nuevos para las columnas por lo cual almacenará el nuevo valor de Length después de cumplir con la función LENGTH (string) lo cual nos retorna el número de caracteres del tipo string, posteriormente mandamos a llamar los campos recién actualizado String, Reversed, Length.</p> <p>Con base al resultado, vemos que el nombre ingresado fue Herong, y nos regreso su inverso, además del número de caracteres.</p>												

**E. Ejemplo de la práctica 5.1 con base de datos
'maquina_expendedora'.**

Procedimiento												
<pre>DROP PROCEDURE IF EXISTS reverse_procedure; DELIMITER // CREATE PROCEDURE reverse_procedure (IN cadena VARCHAR(80), OUT reversed VARCHAR(80), OUT length INTEGER) BEGIN SET reversed = reverse(cadena); SET length = length(cadena); SELECT cadena, reversed, length; END // DELIMITER ;</pre>												
Demostración												
<pre>CALL reverse_procedure((SELECT nombre FROM cliente WHERE id_cliente = 20490688), @Reversed, @Length); SELECT @n:=nombre from cliente where id_cliente = 20490689; CALL ReverseProcedure(@n, @Reversed, @Length);</pre>												
Resultados												
<table><tr><th></th><th>cadena</th><th>reversed</th><th>length</th><th></th><th>@n:=nombre_cliente</th></tr><tr><td>▶</td><td>Ashlee Durham</td><td>mahruD eelhsA</td><td>13</td><td>▶</td><td>Hashim Finley</td></tr></table>		cadena	reversed	length		@n:=nombre_cliente	▶	Ashlee Durham	mahruD eelhsA	13	▶	Hashim Finley
	cadena	reversed	length		@n:=nombre_cliente							
▶	Ashlee Durham	mahruD eelhsA	13	▶	Hashim Finley							
Explicación de resultados												
<p>Este procedimiento funciona igual al anterior, solo con la peculiaridad que maneja la base de datos del proyecto ‘maquina_expendedora’, llamamos al procedimiento de dos maneras, la primera manera es dar el nombre a la variable String mediante una consulta que de un valor escalar, y la segunda manera es igual con una consulta, pero creando una variable de entorno. De igual forma, ambos procedimientos nos invirtieron el nombre proporcionado y nos proporcionaron la longitud de la cadena</p>												

F. Práctica 5.2 sobre procedimientos almacenados con variables.

Competencia:

Utilizar parámetros y variables en los SP.

Introducción:

Dentro de un procedimiento almacenado(SP) se pueden crear variables locales utilizando la sentencia DECLARE , la visibilidad de ellas es dentro del bloque BEGIN ... END donde está declarado. Puede usarse en bloques anidados excepto aquéllos que declaran una variable con el mismo nombre. La sintaxis es la siguiente:

DECLARE variable_name datatype(size) DEFAULT default_value;

- **variable_name** Especifica el nombre de la variable desde de la palabra reservada DECLARE. El nombre debe apegarse a las reglas de nomenclatura de MYSQL para nombres de columna de las tablas
- **datatype(size)** Especificar el tipo de variable y su tamaño. Puede ser cualquiera definido por MYSQL tales como INT, VARCHAR, DATETIME.
- **default_value** Cuando se declara la variable, el valor inicial es NULL, este puede cambiarse utilizando DEFAULT.

Procedimiento								
<pre>DROP PROCEDURE IF EXISTS Invertir_Nombrecliente; DELIMITER // CREATE PROCEDURE Invertir_Nombrecliente (IN cliente int) BEGIN DECLARE nombre_cliente varchar(20); SELECT nombre INTO nombre_cliente from clientes WHERE noCliente=cliente; CALL ReverseProcedure(nombre_cliente, @Reversed, @Length); UPDATE clientes SET nombre=@Reversed WHERE nocliente=cliente; END // DELIMITER ;</pre>								
Demostración								
<pre>call Invertir_Nombrecliente (10024); SELECT nombre from clientes WHERE noCliente=10024; /*10024 es el valor que fue enviado al SP */</pre>								
Resultados								
<table><tr><td></td><td>String</td><td>Reversed</td><td>Length</td></tr><tr><td>►</td><td>HUGO ALEJANDRO</td><td>ORDNAJELA OGUH</td><td>14</td></tr></table>		String	Reversed	Length	►	HUGO ALEJANDRO	ORDNAJELA OGUH	14
	String	Reversed	Length					
►	HUGO ALEJANDRO	ORDNAJELA OGUH	14					
Explicación de resultados								
<p>Se crea un nuevo procedimiento llamado Invertir_NombreCliente la cual cuenta con una variable llamada cliente del tipo entero con parámetros de entrada comienza el procedimiento se declara una variable local llamada nombre_cliente, se procede a realizar una validación para ver si el id_cliente existe. Posteriormente se llama procedimiento anteriormente creado ReverseProcedure con las variables nombre_cliente, @Reversed, @Length y actualiza en la tabla clientes y especifica nombre=@Reversed siempre y cuando nocliente=cliente. Con base a los resultados, podemos ver que la actualización realizada a la tabla resultó satisfactoria, ya que mediante una consulta especificando el número de cliente que afectamos, vimos que realmente sí se invirtió dicho nombre del cliente.</p>								

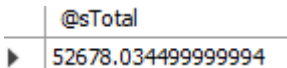
Procedimiento								
<pre>DROP PROCEDURE IF EXISTS Invertir_Nombrecliente2; DELIMITER // CREATE PROCEDURE Invertir_Nombrecliente2 (IN cliente int) BEGIN DECLARE nombre_cliente varchar(20); DECLARE inverso varchar(20); SELECT nombre INTO nombre_cliente from clientes WHERE noCliente=cliente; CALL ReverseProcedure(nombre_cliente, inverso, @Length); UPDATE clientes SET nombre= inverso WHERE nocliente=cliente; END// DELIMITER ;</pre>								
Demostración								
<pre>select * from clientes where nocliente=10024; Call Invertir_Nombrecliente2('10024'); select @Reversed; select inverso;</pre>								
Resultados								
<div><table><tr><th>String</th><th>Reversed</th><th>Length</th></tr><tr><td>ORDNAJELA OGUIH</td><td>HUGO ALEJANDRO</td><td>14</td></tr></table><div><table><tr><th>@Reversed</th></tr><tr><td>HUGO ALEJANDRO</td></tr></table></div></div>	String	Reversed	Length	ORDNAJELA OGUIH	HUGO ALEJANDRO	14	@Reversed	HUGO ALEJANDRO
String	Reversed	Length						
ORDNAJELA OGUIH	HUGO ALEJANDRO	14						
@Reversed								
HUGO ALEJANDRO								
Explicación de resultados								
<p>Básicamente, este procedimiento funciona igual que el anterior, con la diferencia que este no maneja variables de entorno, por lo que la declaración de la variable del reverso del nombre es local, de igual forma, modifica la tabla del cliente; cambiando el nombre por el mismo nombre pero al revés.</p>								

**G. Ejemplo de la práctica 5.2 con base de datos
'maquina_expendedora'.**

Procedimiento								
<pre>DELIMITER // CREATE PROCEDURE invertir_nombre_cliente (IN cliente INTEGER) BEGIN DECLARE nombre_cliente VARCHAR(20); SELECT cliente.nombre_cliente INTO nombre_cliente FROM cliente WHERE id_cliente = cliente; CALL reverse_procedure(nombre_cliente, @reversed, @length); UPDATE cliente SET cliente.nombre_cliente = @reversed WHERE id_cliente = cliente; END // DELIMITER ;</pre>								
Demostración								
<pre>CALL invertir_nombre_cliente(20490687);</pre>								
Resultados								
<table><tr><th></th><th>cadena</th><th>reversed</th><th>length</th></tr><tr><td>▶</td><td>snilloC haraS</td><td>Sarah Collins</td><td>13</td></tr></table>		cadena	reversed	length	▶	snilloC haraS	Sarah Collins	13
	cadena	reversed	length					
▶	snilloC haraS	Sarah Collins	13					
Explicación de resultados								
<p>Utiliza la base de datos ‘maquina_expendedora’. Este procedimiento recibe un número de cliente y mediante el procedimiento de reverse_procedure creado anteriormente, invierte el nombre del cliente y lo almacena en la variable de entorno @reversed, posteriormente actualiza el cliente con base a lo obtenido en @reversed. En el resultado podemos observar la inversión del nombre además de mostrar la longitud de la cadena en función del nombre del cliente.</p>								

Procedimiento								
<pre>CREATE PROCEDURE invertir_nombre_cliente_2 (IN cliente INTEGER) BEGIN DECLARE nombre_cliente VARCHAR(20); DECLARE inverso VARCHAR(20); SELECT cliente.nombre_cliente INTO nombre_cliente FROM cliente WHERE id_cliente = cliente; CALL reverse_procedure(nombre_cliente, inverso, @length); UPDATE cliente SET cliente.nombre_cliente = inverso WHERE id_cliente = cliente; END \$\$ DELIMITER ;</pre>								
Demostración								
<pre>CALL invertir_nombre_cliente_2(20490689);</pre>								
Resultados								
<table><tr><th></th><th>cadena</th><th>reversed</th><th>length</th></tr><tr><td>►</td><td>Hashim Finley</td><td>yelniF mihsaH</td><td>13</td></tr></table>		cadena	reversed	length	►	Hashim Finley	yelniF mihsaH	13
	cadena	reversed	length					
►	Hashim Finley	yelniF mihsaH	13					
Explicación de resultados								
<p>Básicamente, este procedimiento realiza la misma función que el anterior a diferencia que este no maneja variables de entorno para almacenar el nombre del cliente invertido, si no una variable local, que está denotada como nombre_cliente.</p>								

H. Práctica 5.3 sobre procedimientos almacenados con parámetros de entrada y salida.

Procedimiento
<pre> DELIMITER // CREATE PROCEDURE acumuladoVentaArticulo(INOUT subtotal double, IN id_producto INT, IN cantidad INT) BEGIN SELECT PrecioEuros into @incremento_precio FROM Articulos WHERE NoArticulo = id_producto; SET subtotal =subtotal + cantidad * @incremento_precio; END // DELIMITER ; </pre>
Demostración
<pre> SET @sTotal = 0; Call acumuladoVentaArticulo(@sTotal,5,3); Call acumuladoVentaArticulo(@sTotal,12,4); Call acumuladoVentaArticulo(@sTotal,25,7); Call acumuladoVentaArticulo(@sTotal,76,1); Select @sTotal; </pre>
Resultados

Explicación de resultados
<p>Este procedimiento maneja tres parámetros, uno bilateral llamado subtotal, otro de entrada llamado id_producto, y por último otro de entrada, llamado cantidad, dentro del procedimiento se obtiene el precio en euros de artículo que nosotros proporcionamos y lo almacena en la variable de entorno incremento_precio. Posteriormente, se va concatenando en la variable subtotal el producto de la cantidad adquirida por el precio del producto.</p> <p>Con base a los resultados, podemos observar que primeramente el total empieza en cero, conforme vayamos llamando la función con la cantidad de productos determinados que queremos comprar, se va concatenando en la variable total, al final, observamos el total a pagar gracias a la sumatoria de lo obtenido en las llamadas del procedimiento.</p>

I. Ejemplo de la práctica 5.3 con base de datos
'maquina_expendedora'.

Procedimiento						
<pre>DROP PROCEDURE IF EXISTS acumulado_venta_producto; DELIMITER \$\$ CREATE PROCEDURE acumulado_venta_producto (INOUT subtotal DOUBLE, IN id_maquina INT, IN id_producto INT, IN cantidad INT) BEGIN SELECT precio_unitario INTO @precio FROM inventario_maquina WHERE inventario_maquina.id_producto = id_producto AND inventario_maquina.id_maquina = id_maquina; SET subtotal = subtotal + cantidad * @precio; END \$\$ DELIMITER ;</pre>						
Demostración						
<pre>SET @total = 0; CALL acumulado_venta_producto(@total, 4010, 159001, 2); CALL acumulado_venta_producto(@total, 4010, 159002, 6); CALL acumulado_venta_producto(@total, 4010, 159003, 1); CALL acumulado_venta_producto(@total, 4010, 159003, 1);</pre>						
Resultados						
<table><tr><td></td><td>Puntos:</td><td>@total</td></tr><tr><td>►</td><td>Puntos:</td><td>384</td></tr></table>		Puntos:	@total	►	Puntos:	384
	Puntos:	@total				
►	Puntos:	384				
Explicación de resultados						
<p>Básicamente, funciona igual que el anterior procedimiento explicado, con la excepción de que está manejando la tabla de la base de datos del proyecto ‘maquina_expendedora’, además de tener como parámetro de entrada extra la identificación de la máquina, ahora el resultado no lo muestra en puntos, y da la sumatoria total obtenida gracias al llamado de los procedimientos con los parámetros indicados.</p>						

J. Práctica 5.4 sobre procedimientos almacenados con estructuras de control.

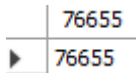
Competencia:

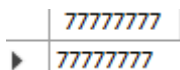
Utilizar estructuras de control en los SP


Introducción:

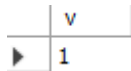
Al igual que cualquier lenguaje de programación, MYSQL cuenta con estructuras de control de flujo que pueden ser utilizadas en los procedimientos almacenados (SP). entre ellas se tienen a:

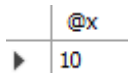
- IF Statement
- CASE Statement
- LOOP Statement
- REPEAT Statement
- WHILE Statement
- ITERATE Statement
- LEAVE Statement

Procedimiento
<pre> delimiter // CREATE procedure miProc(IN p1 int) begin declare miVar int; SET miVar = p1 +1 ; IF miVar = 12 then SELECT 55555; else SELECT 76655; end IF; end; // delimiter ; </pre>
Demostración
<pre>CALL miProc(12);</pre>
Resultados
 <p>The screenshot shows a single row with the value 76655. There is a small arrow icon to the left of the value.</p>
Explicación de resultados
<p>Se crea un procedimiento llamado MiProc con una variable llamada p1 que es de tipo entero y tiene parámetros de entrada, comienza el procedimiento y se declara una variable local llamada MiVar del tipo entero, La cláusula SET va incrementando la variable MiVar, si la variable miVar es igual a 12 entonces nos muestra 55555, en caso contrario nos mostraría 76655. Es por eso que nos muestra 76655 ya que no se cumplió la condición de la condicional IF.</p>

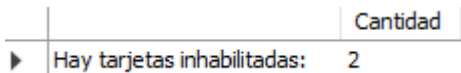
Procedimiento
<pre> delimiter // CREATE procedure miProc2 (IN p1 int) begin declare var int ; set var = p1 +2 ; case var when 2 then SELECT (66666); when 3 then SELECT (4545665); else SELECT (7777777); end case; end; //</pre>
Demostración
<pre>CALL miProc2(12);</pre>
Resultados

Explicación de resultados
<p>Se crea un procedimiento llamado MiProc2 con una variable llamada p1 del tipo entero como parámetro de entrada, comienza el procedimiento y se declaró una variable local la cual se llama var del tipo entero, La cláusula SET incrementa la variable var de dos en dos, en caso de que la variable var valga 2 nos mostrará el resultado “66666” en caso que el valor de la variable var valga 3 nos mostrará el resultado “4545665” en caso de que ninguna de estos casos se cumplan nos mostrará el siguiente resultado “7777777”, como el número ingresado fue 12, el resultado no cumplió con ningún caso y terminó mostrando “7777777” .</p>

Procedimiento		
<pre>CREATE procedure p14() begin declare v int; set v = 0; while v < 5 do SET v = v + 1 ; end while; SELECT v; end; // delimiter ;</pre>		
Demostración		
<pre>CALL p14(); show procedure status;</pre>		
Resultados		
 <table border="1"> <thead> <tr> <th>v</th> </tr> </thead> <tbody> <tr> <td>5</td> </tr> </tbody> </table>	v	5
v		
5		
Explicación de resultados		
<p>Se crea un procedimiento llamado P14, comienza el procedimiento y se declara la variable local V de tipo entero. La cláusula SET inicializa la variable V, comienza el ciclo while se compara la variable V y si es menor igual a 5. La cláusula SET va incrementando en uno a la variable V. El ciclo se repite hasta que V sea igual a 5, y nos da ese valor como resultado.</p>		

Procedimiento		
<pre> delimiter // CREATE procedure p15() begin declare v int; set v = 20; repeat SET v = v - 1; until v <= 1 end repeat; SELECT v; end; // delimiter ; </pre>		
Demostración		
<pre>CALL p15();</pre>		
Resultados		
 <table border="1"> <thead> <tr> <th>v</th> </tr> </thead> <tbody> <tr> <td>1</td> </tr> </tbody> </table>	v	1
v		
1		
Explicación de resultados		
<p>Este procedimiento no recibe parámetros de entrada, se inicializa la variable local V en 20 y se va decrementando de uno en uno hasta que el valor de la variable V sea menor o igual a 1, en ese caso se sale del ciclo y nos muestra el valor de la variable V, que este caso fue 1 debido a que fue el valor que permitió salirnos del bucle.</p>		

Procedimiento		
<pre> CREATE PROCEDURE doiterate(p1 INT) BEGIN label1: LOOP SET p1 = p1 + 1; IF p1 < 10 THEN ITERATE label1; END IF; LEAVE label1; END LOOP label1; SET @x = p1; END\$ delimiter ; </pre>		
Demostración		
<pre> CALL doiterate(); SELECT @x </pre>		
Resultados		
 <table border="1"> <thead> <tr> <th>@x</th></tr> </thead> <tbody> <tr> <td>10</td></tr> </tbody> </table>	@x	10
@x		
10		
Explicación de resultados		
<p>Este procedimiento tiene la variable de tipo entero p1 de entrada, y se inicia una etiqueta de bucle con el nombre de label1, se irá incrementando p1 y mientras que p1 sea menor a 10 será iterando lo mismo de la incrementación, ya que se cumpla la condición se saldrá del bucle e imprimirá el valor de la variable de entorno x que obtiene el valor de p1. Finalmente, nos muestra el resultado el cual siempre será 10 porque indica el final de nuestra condición.</p>		

**K. Ejemplo de la práctica 5.4 con base de datos
'maquina_expendedora'.**

Procedimiento
<pre> DROP PROCEDURE IF EXISTS verificar_tarjetas_inhabilitadas; DELIMITER // CREATE PROCEDURE verificar_tarjetas_inhabilitadas (IN id_cliente INT) BEGIN DECLARE tarjetas_inhabilitadas INT; IF ((SELECT estatus FROM tarjeta WHERE tarjeta.id_cliente = id_cliente AND estatus = 'Deshabilitada' LIMIT 1) = 'Deshabilitada') THEN BEGIN SELECT count(*) INTO tarjetas_inhabilitadas FROM tarjeta WHERE tarjeta.id_cliente = id_cliente AND estatus = 'Deshabilitada'; SELECT "Hay tarjetas inhabilitadas: " AS "", tarjetas_inhabilitadas AS "Cantidad"; END; ELSE SELECT "No hay tarjetas inhabilitadas."; END IF; END; // DELIMITER ; </pre>
Demostración
<pre> SELECT * FROM tarjeta; CALL verificar_tarjetas_inhabilitadas(20490688); </pre>
Resultados

Explicación de resultados
<p>Este procedimiento se encarga de verificar cuántas tarjetas inhabilitadas el cliente que proporcionamos. El procedimiento tiene como parámetro de entrada el id_cliente y primeramente se verifica si el cliente tiene tarjetas deshabilitadas, en caso de sí tener se prosigue a contar cuántas tiene, y por último nos despliega la cantidad de tarjetas deshabilitadas sí tiene y si no tiene, muestra el caso contrario. En este caso se proporcionó el número de cliente 20490688, y nos dice que tiene dos tarjetas deshabilitadas.</p>

Procedimiento

```
CREATE PROCEDURE agregar_impuesto_producto (IN id_producto VARCHAR(8))
BEGIN
    SELECT id_maquina, id_producto, precio_unitario,
    CASE
        WHEN precio_unitario > 35
            THEN precio_unitario * 1.16
            ELSE "N/A"
        END AS "precio_actualizado"
    FROM inventario_maquina
    WHERE inventario_maquina.id_producto = id_producto ;
END; $$
DELIMITER ;
```

Demostración

```
SELECT * FROM inventario_maquina;
CALL agregar_impuesto_producto(159001);
```

Resultados

	id_maquina	id_producto	precio_unitario	precio_actualizado
►	4010	159001	36	41.76
	4011	159001	36	41.76
	4012	159001	36	41.76
	4020	159001	53	61.48

Explicación de resultados

Este procedimiento tiene como parámetro de entrada el id_producto, básicamente lo que realiza el procedimiento es hacer un inventario de ese producto en todas las máquinas donde se encuentra, y si el precio unitario es mayor a 35, se le aplicará un impuesto de 1.16 puntos, en caso contrario, no se le aplicará. Como resultado nos despliega una tabla del inventario con respecto a ese producto, y con su respectivo precio actualizado en caso de que sea mayor a 35.

Procedimiento
<pre> DROP PROCEDURE IF EXISTS costo_total_maquina; DELIMITER // CREATE PROCEDURE costo_total_maquina (IN id_maquina VARCHAR(4)) BEGIN DECLARE precio_total INT; DECLARE precio_obtenido INT; DECLARE cantidad_total INT; DECLARE i INT; SET i = (SELECT id_producto FROM inventario_maquina WHERE inventario_maquina.id_maquina = id_maquina ORDER BY id_producto LIMIT 1); SET precio_total = 0; SET precio_obtenido = 0; SET cantidad_total = 0; WHILE i <= (SELECT id_producto FROM inventario_maquina WHERE inventario_maquina.id_maquina = id_maquina ORDER BY id_producto DESC LIMIT 1) DO SELECT precio_unitario, cantidad INTO precio_obtenido, cantidad_total FROM inventario_maquina WHERE id_producto = i AND inventario_maquina.id_maquina = id_maquina; SET precio_total = precio_total + precio_obtenido * cantidad_total; SET i = i + 1; END WHILE; SELECT id_maquina AS "Máquina", precio_total AS "Costo Total en Puntos"; END; // DELIMITER ; </pre>
Demostración
<pre> SELECT * FROM inventario_maquina; CALL costo_total_maquina('4010'); </pre>
Resultados

	Máquina	Costo Total en Puntos
▶	4010	10251

Explicación de resultados

Este procedimiento nos pide como parámetro de entrada el id_maquina, en el cual nos mostrará el dinero total en puntos de esa máquina en específico, se declara la variable i en el valor menor de nuestra id_producto, será nuestra índice y se irá incrementando para obtener el precio total mediante el producto del precio unitario por la cantidad de producto en ese slot de la máquina, y seguirá con la iteración mientras que i sea menor o igual al valor mayor de id_producto. Finalmente nos muestra el identificador de la máquina junto al costo total en puntos.

Procedimiento

```

DROP PROCEDURE IF EXISTS total_facturado;
DELIMITER //
CREATE PROCEDURE total_facturado()
BEGIN
    DECLARE total_facturado INT;
    DECLARE total_facturado_cliente INT;
    DECLARE i INT;

    SET i = (SELECT id_cliente FROM cliente ORDER BY id_cliente LIMIT 1);
    SET total_facturado = 0;

    REPEAT
        SET total_facturado_cliente = 0;

        SELECT sum(precio_unitario * detalle.cantidad) INTO
total_facturado_cliente
        FROM factura
        INNER JOIN detalle using(id_factura)
        INNER JOIN inventario_maquina using(id_maquina,
id_producto)
        WHERE id_cliente = i
        GROUP BY id_cliente;

        SET total_facturado = total_facturado +
total_facturado_cliente;

        SET i = i + 1;
    
```

```

        UNTIL i >= (SELECT id_cliente FROM cliente ORDER BY id_cliente
DESC LIMIT 1)
        END REPEAT;

        SELECT total_facturado AS 'Total Facturado';
END; //
DELIMITER ;

```

Demostración

```

SELECT *
FROM factura
    INNER JOIN detalle using(id_factura)
    INNER JOIN inventario_maquina using(id_maquina,
id_producto);
CALL total_facturado();

```

Resultados

Total Facturado
19233

Explicación de resultados

Este procedimiento no tiene parámetros de entrada y nos muestra precisamente cuánto dinero en puntos se ha facturado en específico durante todo el tiempo que se ha comprado en las máquinas. Primeramente, se inicializa la variable local `i` con respecto al `id_cliente` mejor, se irá obteniendo el total facturado por cliente, se va sumando en la variable `total_facturado_cliente` y se irá incrementando `i` para desplazarnos de cliente. Finalmente finalizará hasta que `i` sea mayor o igual al identificador de cliente mayor en la tabla, o sea, el último cliente. Como resultado nos muestra el valor escalar correspondiente al total facturado en el sistema en puntos.

Procedimiento

```

DROP PROCEDURE IF EXISTS saldo_total_tarjetas_inhabilitadas;
DELIMITER $$
CREATE PROCEDURE saldo_total_tarjetas_inhabilitadas ()
BEGIN
    DECLARE total INT;
    DECLARE i INT;

    SET total = 0;

```

```

SET i = (SELECT DISTINCT id_cliente FROM tarjeta WHERE estatus =
"Deshabilitada" LIMIT 1);

label_1: LOOP

SET total = total +
(
    SELECT sum(puntos)
    FROM tarjeta
    WHERE estatus = "Deshabilitada"
    AND id_cliente = i
);

SET i = i + 1;

IF i <= (SELECT DISTINCT id_cliente FROM tarjeta WHERE estatus =
"Deshabilitada" ORDER BY id_cliente DESC LIMIT 1) THEN
    ITERATE label_1;
END IF;

LEAVE label_1;
END LOOP label_1;

SELECT total AS "Saldo total en tarjetas deshabilitadas";
END $$
DELIMITER ;

```

Demostración

```

SELECT id_cliente, sum(puntos)
FROM tarjeta
WHERE estatus = "Deshabilitada"
GROUP BY id_cliente;

CALL saldo_total_tarjetas_inhabilitadas();

```

Resultados

	Saldo total en tarjetas deshabilitadas
►	53480

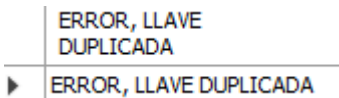
Explicación de resultados

Este procedimiento no tiene parámetros de entrada y nos muestra precisamente cuántos puntos en total se tiene de todas las tarjetas deshabilitadas en el sistema. Primeramente, se inicializa la variable local i con respecto al id_cliente mejor, se


irá obteniendo el la sumatoria de puntos de las tarjetas deshabilitas por cliente, se va sumando en la variable total y se irá incrementando i para desplazarnos de cliente. Finalmente finalizará hasta que i sea mayor o igual al identificador de cliente mayor en la tabla, o sea, el último cliente. Como resultado nos muestra el valor escalar correspondiente al saldo total de las tarjetas deshabilitadas que hay en el sistema.

L. Práctica 5.5 sobre manejadores en procedimientos almacenados.

Procedimiento						
<pre>DROP PROCEDURE IF EXISTS handlerdemo DELIMITER // CREATE PROCEDURE handlerdemo () BEGIN DECLARE CONTINUE HANDLER FOR SQLSTATE '23000' SET @x2 = 1; SET @x = 1; INSERT INTO articulos VALUES (154,"mouse KM-123",200); SET @x = 2; INSERT INTO articulos VALUES (154,"mouse KM-123",200); SET @x = 3; END // DELIMITER ;</pre>						
Demostración						
<pre>SET @x = 0, @x2 = 0; CALL handlerdemo(); SELECT @x, @x2;</pre>						
Resultados						
<table><tr><th></th><th>@x</th><th>@x2</th></tr><tr><td>▶</td><td>3</td><td>1</td></tr></table>		@x	@x2	▶	3	1
	@x	@x2				
▶	3	1				
Explicación de resultados						
<p>Este procedimiento hace uso de los manejadores que en caso de obtener el error de llave primaria duplicado se activa. En primera instancia, se inicializa el valor de la variable de entorno x en 1, se hace una inserción de un artículo que se supone no se encuentra en la tabla articulos, y ahora x vale 2, después se vuelve hacer la misma inserción del artículo antes mencionado, pero como sucede un error, ahora nos lleva al manejador, donde se establece la variable de entorno x2 como 1 y la manera en la que se va abordar el error es que siga el programa, por lo que ahora x vale 3. Al final, si mostramos los valores de las variables observamos que x ahora vale 3 y x2 vale 1, por los motivos anteriormente mencionados.</p>						

Procedimiento
<pre> DROP PROCEDURE IF EXISTS handInsertar; DELIMITER // CREATE PROCEDURE handInsertar(IN ID INT, IN NOMBRE VARCHAR(50), IN PRECIO DOUBLE) BEGIN DECLARE continue HANDLER FOR SQLSTATE '23000' SET @ERROR = 1; SET @ERROR=0; INSERT INTO articulos VALUES (ID,NOMBRE,PRECIO); IF @ERROR=1 THEN SELECT "ERROR, LLAVE DUPLICADA"; END IF; END // DELIMITER ; </pre>
Demostración
<pre> SELECT * FROM articulos; CALL handInsertar(1, "Tarjeta Gráfica", 77.77); </pre>
Resultados
 <p>The screenshot shows a SQL query result window. It contains a single row with the text 'ERROR, LLAVE DUPLICADA'. The text is displayed in a monospaced font, with 'ERROR,' in black and 'LLAVE DUPLICADA' in red. There is a small cursor icon to the left of the text.</p>
Explicación de resultados
<p>En este procedimiento se tienen tres parámetros de entrada; el identificador del artículo, el nombre del artículo y el precio, primero se establece la variable de entorno error igual a 0, se hace la inserción del artículo, pero si llega haber un error al momento de hacer la inserción, se llevará al manejador y se establecerá que la variable error vale ahora 1, entonces se continúa con el procedimiento. Si el error ahora vale 1, se mostrará el mensaje de que se tiene una llave duplicada. Como se observa en el resultado, se quiso hacer un inserción de un artículo con un identificador de artículo que ya existe, y por ende nos dió error, porque no permite que haya llaves primarias duplicadas.</p>

**M. Ejemplo de la práctica 5.5 con base de datos
'maquina_expendedora'.**

Procedimiento
<pre> DROP PROCEDURE IF EXISTS handler_insertar_cliente /* Procedimiento almacenado que hace llamar a manejador cuando se ingresa un identificador de cliente que ya está en la tabla 'cliente' (Duplicado). */ DELIMITER // CREATE PROCEDURE handler_insertar_cliente (IN id_cliente INT, IN id_institucion INT, IN nombre_cliente VARCHAR(64), IN apellido_paterno VARCHAR(64), IN apellido_materno VARCHAR(64)) BEGIN DECLARE EXIT HANDLER FOR SQLSTATE '23000' BEGIN SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT = 'Error. Identificador de cliente duplicado.'; END; INSERT INTO cliente VALUES (id_cliente, id_institucion, nombre_cliente, apellido_paterno, apellido_materno); SELECT "Inserción de cliente concluida satisfactoriamente." AS ""; END // DELIMITER ; </pre>
Demostración
<pre> SELECT * FROM cliente; CALL handler_insertar_cliente(20490687, 1010, "Luis", "Barba", "Navarro"); </pre>
Resultados
<div>  47 18:42:20 CALL handler_insertar_cliente(20490687, 1010, "Luis", "Barba", "Navarro") </div> <div>Error Code: 1644. Error. Identificador de cliente duplicado.</div>
Explicación de resultados

Este procedimiento funciona igual que el anterior, pero con la peculiaridad que se está ejecutando desde la base de datos 'maquina_expendedora'. Tiene como parámetros de entrada: el identificador de cliente, el identificador de la institución, el nombre, apellido paterno, y apellido materno del cliente. Se hace la inserción con esas variables proporcionadas pero si llega haber un error de manda al manejador el cuál tiene como instrucción mandarnos un mensaje de error en el log diciendo que el identificador del cliente se encuentra duplicado, después finaliza el programa. Si llega a finalizar de forma exitosa, nos muestra en mensaje que la inserción del cliente terminó de manera exitosa. Con base a los resultados vemos que se quiso hacer una inserción de un cliente que ya existe, es por eso que nos manda el error de identificador de cliente duplicado, por lo tanto no se hizo la inserción.

Procedimiento

```
DROP PROCEDURE IF EXISTS handler_asignar_tarjeta;
DELIMITER //
CREATE PROCEDURE handler_asignar_tarjeta
(
    IN id_tarjeta INT,
    IN id_cliente INT
)
BEGIN
    -- Tarjeta duplicada.
    DECLARE EXIT HANDLER
    FOR 1062
    SELECT 'Identificador de tarjeta duplicado.' AS "";

    -- Cliente no encontrado.
    DECLARE EXIT HANDLER
    FOR 1452
    SELECT 'Identificador de cliente no encontrado.' AS "";

    INSERT tarjeta VALUES (id_tarjeta, id_cliente, CURRENT_DATE(), 0,
    'Habilitada');
    SELECT 'Tarjeta asignada satisfactoriamente.';
END //
DELIMITER ;
```

Demostración

```
SELECT * FROM tarjeta;
CALL handler_asignar_tarjeta(10, 20490999);
```

Resultados	
	<div> <div></div> <div>Identificador de cliente no encontrado.</div> </div> <div> <div></div> <div>Identificador de tarjeta duplicado.</div> </div>
Explicación de resultados	
<p>Este procedimiento nos permite asignar una tarjeta a un cliente. Como parámetros de entrada está el identificador de la tarjeta, el cuál nosotros proporcionamos y el identificador del cliente, con esos datos se hace la asignación de la tarjeta. Puede haber dos posibles errores; si el cliente que proporcionamos no existe, nos mostrará en pantalla que el identificador de cliente no fue encontrado, como se muestra en resultados. Por otro lado, si nosotros ingresamos un identificador de tarjeta que ya está utilizado, nos mostrará este mensaje de error. De igual forma, al presentarse estos errores automáticamente terminará el procedimiento y no se completará la inserción.</p>	

N. Práctica 5.6 sobre disparadores en procedimientos almacenados.

Procedimiento																								
<pre>DROP TRIGGER IF EXISTS actualiza_salario_deptoAI; delimiter // CREATE TRIGGER actualiza_salario_deptoAI AFTER INSERT on empleado FOR EACH ROW BEGIN IF new.idDepto is not null THEN update departamentos set total_salario=total_salario+new.salario where departamentos.IdDepto=new.IdDepto; END IF; END;// delimiter ;</pre>																								
Demostración																								
<pre>INSERT INTO empleado (IdEmpleado, nombre, salario, IdDepto) VALUES ('100', 'CARLOS LOPEZ', '1900', '1'); INSERT INTO empleado (IdEmpleado, nombre, salario, IdDepto) VALUES ('200', 'JUAN PEREZ', '1800', '2'); INSERT INTO empleado (IdEmpleado, nombre, salario, IdDepto) VALUES ('300', 'MARIA GOMEZ', '2000', '3'); INSERT INTO empleado (IdEmpleado, nombre, salario, IdDepto) VALUES ('400', 'GABRIELA ARENAS', '1300', '4'); select * from departamentos;</pre>																								
Resultados																								
<table><tr><td></td><td>IdDepto</td><td>nombre</td><td>total_salario</td></tr><tr><td>▶</td><td>1</td><td>produccion</td><td>1900</td></tr><tr><td></td><td>2</td><td>ventas</td><td>1800</td></tr><tr><td></td><td>3</td><td>almacen</td><td>2000</td></tr><tr><td></td><td>4</td><td>administrativo</td><td>1300</td></tr><tr><td>*</td><td>NULL</td><td>NULL</td><td>NULL</td></tr></table>		IdDepto	nombre	total_salario	▶	1	produccion	1900		2	ventas	1800		3	almacen	2000		4	administrativo	1300	*	NULL	NULL	NULL
	IdDepto	nombre	total_salario																					
▶	1	produccion	1900																					
	2	ventas	1800																					
	3	almacen	2000																					
	4	administrativo	1300																					
*	NULL	NULL	NULL																					
Explicación de resultados																								
<p>Este disparador es activado después de una inserción en la tabla empleado, lo que hará es sumarle al total de salario del departamento en la tabla ‘departamento’, el salario del empleado recién agregado. Como podemos observar se realizaron cuatro inserciones, donde el disparador fue activado y en la tabla ‘departamentos’ se vio reflejado el aumento del total de salario por departamento. Por lo que podemos observar que el resultado coincide con el salario ingresado en las inserciones.</p>																								

Procedimiento

```
DROP TRIGGER IF EXISTS actualiza_salario_deptoAU;
delimiter //
CREATE TRIGGER actualiza_salario_deptoAU AFTER UPDATE on empleado
FOR EACH ROW
BEGIN
    IF old.idDepto is not null THEN
        update departamentos set
total_salario=total_salario-old.salario
        where departamentos.IdDepto=old.IdDepto;
    END IF;
    IF new.idDepto is not null THEN
        update departamentos set
total_salario=total_salario+new.salario
        where departamentos.IdDepto=new.IdDepto;
    END IF;
END;//
delimiter ;
```

Demostración

```
UPDATE empleado SET salario = '2000' WHERE (IdEmpleado = '100');
UPDATE empleado SET salario = '3300' WHERE (IdEmpleado = '300');
```

Resultados

	IdDepto	nombre	total_salario
▶	1	produccion	2000
	2	ventas	1800
	3	almacen	3300
	4	administrativo	0
*	NULL	NULL	NULL

Explicación de resultados

Este disparador se activa después de realizar una actualización de un registro en la tabla 'empleado', donde si nosotros modificamos el salario del cliente, en el disparador pasan dos cosas; la primera es que el saldo anterior o sea el saldo viejo es restado del total de salario por departamento de la tabla 'departamento', y la segunda es que el salario nuevo es sumado al total de salario por departamento. Como podemos observar en los resultados, el salario del empleado identificado con 100 y 300 fue modificado, y se reflejó en la tabla 'departamento'; siendo el departamento de producción y almacén los afectados en cuestión del salario total.

Procedimiento

```
DROP TRIGGER IF EXISTS actualiza_salario_deptoAD;
delimiter //
CREATE TRIGGER actualiza_salario_deptoAD AFTER DELETE on empleado
FOR EACH ROW
BEGIN
IF old.idDepto is not null THEN
update departamentos set total_salario=total_salario-old.salario
where departamentos.IdDepto=old.IdDepto;
END IF;
END;//
delimiter ;
```

Demostración

```
delete from empleado where (IdEmpleado = '100');
delete from empleado where (IdEmpleado = '400');


select * from departamentos;
```

Resultados

	IdDepto	nombre	total_salario
▶	1	produccion	0
	2	ventas	1800
	3	almacen	3300
	4	administrativo	0
*	NULL	NULL	NULL

Explicación de resultados

Este disparador se activa después de realizar una eliminación de registro en la tabla 'empleado'. Cuando nosotros eliminamos un empleado, el saldo asociado a este será eliminado también del total de salario por departamento. En este caso fueron eliminados los empleados con identificación 100 y 400 respectivamente. Por lo que en la tabla 'departamento' podemos observar que el saldo total de los departamentos es igual a 0.

Procedimiento
<pre>drop TRIGGER if exists valida_sueldo_BU; delimiter // CREATE TRIGGER valida_sueldo_BU BEFORE update on empleado FOR EACH ROW BEGIN IF new.salario<=0 THEN SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT = 'Error en sueldo'; END IF; END;// delimiter ;</pre>
Demostración
<pre>UPDATE empleado SET salario = -10 WHERE (IdEmpleado = '200'); select * from empleado; select * from departamentos;</pre>
Resultados
<div>  93 19:02:04 UPDATE empleado SET salario = -10 WHERE (IdEmpleado = '200') </div> <div>Error Code: 1644. Error en sueldo</div>
Explicación de resultados
<p>Este procedimiento se activa antes de realizar una actualización en la tabla 'empleado', en el caso que nosotros ingresemos un salario menor o igual a 0 nos mandará un error y no nos dejará hacer la actualización del salario. Como podemos observar, se trató de actualizar el salario del empleado con identificación 200, con un salario negativo, y el disparador nos arrojó el error de sueldo, como se observa en el resultado.</p>

**O. Ejemplo de la práctica 5.6 con base de datos
'maquina expendedora'.**

Procedimiento					
<pre> DROP TRIGGER IF EXISTS asignar_tarjeta_AI; /* Disparador que se activa cuando creamos un cliente nuevo, nos permite crear y asignar una tarjeta al nuevo cliente con quince puntos de regalo. */ DELIMITER // CREATE TRIGGER asignar_tarjeta_AI AFTER INSERT ON cliente FOR EACH ROW BEGIN DECLARE final_id_tarjeta INT; SET final_id_tarjeta = (SELECT id_tarjeta FROM tarjeta ORDER BY id_tarjeta DESC LIMIT 1); IF NEW.id_cliente IS NOT NULL THEN INSERT INTO tarjeta VALUES ((final_id_tarjeta) + 1, NEW.id_cliente, CURRENT_DATE(), 15, "Habilitada"); END IF; END // DELIMITER ; </pre>					
Demostración					
<pre> SELECT * FROM cliente; SELECT * FROM tarjeta; DELETE FROM cliente WHERE id_cliente = 1; INSERT INTO cliente (id_cliente, nombre_cliente, apellido_paterno, apellido_materno) VALUES (3, "Rodrigo", "Barba", "Navarro"); </pre>					
Resultados					
	3	NULL	Rodrigo	Barba	Navarro
	20490687	1010	Sarah Collins	Romero	Spears
	20490688	1010	Ashlee Durham	Nash	Patton

	id_tarjeta	id_cliente	fecha_expedicion	puntos	estatus
▶	10020664	3	2022-12-10	15	Habilitada
	10020663	2	2022-12-07	999	Habilitada
	10020662	1	2022-12-07	15	Habilitada

Explicación de resultados					
<p>Este disparador fue creado desde la base de datos de 'maquina_expendedora ' y se activa cuando creamos un cliente nuevo, nos permite crear y asignar una tarjeta al nuevo cliente con quince puntos de regalo. Como podemos apreciar, dentro del disparador primero se obtiene el último id_tarjeta que se tiene para después asignarle la tarjeta al cliente con el incremento en uno del último id_tarjeta. En los resultados, se observa que se ingresó un cliente al sistema y se le asignó la tarjeta satisfactoriamente con 15 puntos de regalo.</p>					

Procedimiento
<pre> DROP TRIGGER IF EXISTS validar_saldo_tarjeta_AD; /* Disparador que valida el saldo de la tarjeta para que no sea menor o igual a 0 o en su defecto, que sobrepase 9999 por desbordamiento, esto se realizará antes de hacer una actualización sobre la tabla 'tarjeta'. */ DELIMITER // CREATE TRIGGER validar_saldo_tarjeta_AD BEFORE UPDATE ON tarjeta FOR EACH ROW BEGIN IF NEW.puntos <= 0 THEN SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT = 'Error. Saldo ingresado no es válido.'; END IF; IF NEW.puntos >= 9999 THEN SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT = 'Error. Saldo ingresado supera el límite de lo establecido.'; END IF; END // DELIMITER ; </pre>
Demostración
<pre> UPDATE tarjeta </pre>

```

SET puntos = puntos + (9999)
WHERE id_cliente = 20490687 AND estatus = "Habilitada";

UPDATE tarjeta
SET puntos = -1
WHERE id_cliente = 20490687 AND estatus = "Habilitada";

```

Resultados

❌ 101 19:11:13 UPDATE tarjeta SET puntos = puntos + (9999) WHERE id_cliente = 20490687 AND estatus = "Habilitada"

Error Code: 1644. Error. Saldo ingresado supera el límite de lo establecido.

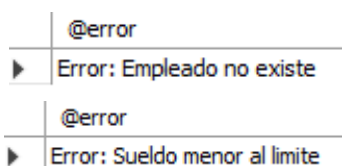
❌ 104 19:12:08 UPDATE tarjeta SET puntos = -1 WHERE id_cliente = 20490687 AND estatus = "Habilitada"

Error Code: 1644. Error. Saldo ingresado no es válido.

Explicación de resultados

Este disparador fue creado desde la base de datos de 'maquina_expendedora', y valida el saldo de la tarjeta para que no sea menor o igual a 0 o en su defecto, que sobrepase 9999 por desbordamiento, esto se realizará antes de hacer una actualización sobre la tabla 'tarjeta'. En el primer caso se intentó actualizar el saldo de cliente con identificación 20490687 para que sea mayor a 9999, nos arrojó un error donde nos dice que el saldo supera lo establecido. En otra instancia, se intentó actualizar el saldo de ese mismo cliente pero en saldo negativo, nuevamente no nos permitió porque el saldo es menor a cero. Al final, estas actualizaciones no se finalizaban y no modificaban nada en la tabla 'tarjeta'.

P. Práctica 5.7 sobre disparadores en procedimientos almacenados con conceptos mixtos.

Procedimiento
<pre>drop procedure if exists actualizarSueldo; DELIMITER // CREATE PROCEDURE actualizarSueldo (in idEmp int, in sueldo double, out error varchar(30)) BEGIN DECLARE CONTINUE HANDLER FOR 1644 SET error = 'Error: Sueldo menor al limite'; select count(*) into @x from empleado where idEmpleado = idEmp; if @x=1 then set error = 'Actualizacion Exitosa'; update empleado set salario=sueldo where idEmpleado=idEmp; else set error = 'Error: Empleado no existe'; end if; END // DELIMITER ;</pre>
Demostración
<pre>set @error=''; call actualizarSueldo (200, -1, @error) ; select @error;</pre>
Resultados

Explicación de resultados
<p>Este procedimiento hace uso de manejadores, y del disparador que anteriormente habíamos creado para evitar saldos negativos. Primeramente, se tiene como parámetro de entrada el identificador del empleado y el sueldo del empleado, y como parámetro de salida, el mensaje de error. Primero se verifica si el cliente existe mediante la consulta con count(), si existe se procede a hacer la actualización, si no, nos arroja el error que el empleado no existe. En caso de</p>

existir, nos actualiza el salario, pero si este es saldo negativo, se activa el disparador que habíamos creado anteriormente y nos manda al manejador, este manejador establecerá el valor del mensaje de error a sueldo menor al límite, finalmente seguirá con la ejecución del procedimiento. En este caso, se hicieron dos pruebas, la primera fue actualizar el sueldo de cliente con uno que no existe, es por eso que nos arrojó el error de no existencia del empleado, y en el otro caso, se actualizó con una cantidad de saldo negativa, la cual resultó fallida igualmente, porque no podemos ingresar saldo menor a cero.

Procedimiento		
<pre> drop procedure if exists actualizarSueldoNombre; delimiter // CREATE PROCEDURE actualizarSueldoNombre (in idEmp int, in sueldo double, in nombreCompleto varchar(15),out error varchar(30)) BEGIN DECLARE CONTINUE HANDLER FOR SQLSTATE '45000' begin set error = 'Error: Sueldo menor al limite'; Rollback; end; select count(*) into @x from empleado where idEmpleado = idEmp; if @x=1 then start Transaction; set error = 'Actualizacion Exitosa'; update empleado set nombre=nombreCompleto where idEmpleado=idEmp; update empleado set salario=sueldo where idEmpleado=idEmp; commit; else set error = 'Error: Empleado no existe'; end if; END // delimiter ; </pre>		
Demostración		
<pre> set @error=''; call actualizarSueldoNombre(700, 0, "Juan Perez", @error) ; select @error; </pre>		
Resultados		
<table> <tr> <td>@error</td></tr> <tr> <td>Error: Empleado no existe</td></tr> </table>	@error	Error: Empleado no existe
@error		
Error: Empleado no existe		

<div> <div>@error</div> <div>Error: Sueldo menor al limite</div> </div>
Explicación de resultados
<p>Este procedimiento hace lo mismo que el anterior, a diferencia que este también actualiza el nombre del empleado.</p> <p>Este procedimiento hace uso de manejadores, y del disparador que anteriormente habíamos creado para evitar saldos negativos. Primeramente, se tiene como parámetro de entrada el identificador del empleado y el sueldo del empleado, y como parámetro de salida, el mensaje de error. Primero se verifica si el cliente existe mediante la consulta con count(), si existe se procede a hacer la actualización, si no, nos arroja el error que el empleado no existe. En caso de existir, nos actualiza el salario y el nombre del empleado, pero si este es saldo negativo, se activa el disparador que habíamos creado anteriormente y nos manda al manejador, este manejador establecerá el valor del mensaje de error a sueldo menor al límite, finalmente seguirá con la ejecución del procedimiento.</p> <p>En este caso, se hicieron dos pruebas, la primera fue actualizar el sueldo de cliente con uno que no existe, es por eso que nos arrojó el error de no existencia del empleado, y en el otro caso, se actualizó con una cantidad de saldo negativa, la cual resultó fallida igualmente, porque no podemos ingresar saldo menor a cero.</p>

Procedimiento
<pre> DELIMITER // CREATE function f_actualizarSueldo (idEmp int, sueldo double) RETURNS VARCHAR(100) DETERMINISTIC BEGIN DECLARE MENSAJE varchar(100); DECLARE CANT INT; DECLARE CONTINUE HANDLER FOR 1644 SET MENSAJE = 'Error: Sueldo menor al limite'; select count(*) INTO CANT from empleado where idEmpleado = idEmp LIMIT 1; if CANT=1 then set MENSAJE = 'Actualizacion Exitosa'; Update empleado set salario=sueldo where idEmpleado=idEmp; else set MENSAJE = 'Error: Empleado no existe'; end if; RETURN MENSAJE; END //</pre>

DELIMITER ;

Demostración

```
SET @error = 0;
SET @error = (f_actualizarSueldo(1234, 7777));
select @error;

SET @error = 0;
SET @error = (f_actualizarSueldo(1, -1));
select @error;
```

Resultados

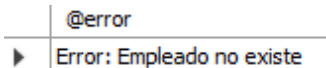
@error
▶ Error: Empleado no existe
@error
▶ Error: Sueldo menor al limite

Explicación de resultados

Esta función tiene como parámetros de entrada el identificador de empleado y el sueldo el cual se desea asignar, y tiene como retorno cualquier cadena de 100 caracteres. Primero se verifica si el empleado existe, si no existe se cambia el mensaje de error y se retorna y finaliza la función. Caso contrario, realizará la actualización del salario, si este saldo es menor a 0, entonces el disparador se activa y nos manda al manejador, el cual cambiará el mensaje de error por el error presente. Se hicieron dos casos de prueba, el primero donde el empleado no existe, nos manda el error correspondiente, y el segundo caso es para un empleado que sí existe, pero el salario es negativo, por lo que nos manda el error que es menor al límite establecido.

La principal diferencia que se percibió de las funciones y los procedimientos almacenados es que una función tiene un tipo devuelto y devuelve un valor y que también se puede llamar a una función mediante una instrucción SELECT. En este caso se pudo asignar el valor de la variable de entorno 'error' con lo que retorna la función, que es el mensaje de error, por lo que puede llegar a ser bastante versátil en cuestión de obtención de datos.

**Q. Ejemplo de la práctica 5.7 con base de datos
'maquina_expendedora'.**

Procedimiento
<pre> drop procedure if exists actualizarSaldoNombre; delimiter // CREATE PROCEDURE actualizarSaldoNombre (in id_cliente int, in nuevos_puntos int, in nombreCompleto varchar(64),out error varchar(30)) BEGIN DECLARE CONTINUE HANDLER FOR SQLSTATE '45000' begin set error = 'Error: Sueldo menor al limite'; Rollback; end; select count(*) into @x from tarjeta where tarjeta.id_cliente = id_cliente and estatus = "Habilitada"; if @x=1 then start Transaction; set error = 'Actualizacion Exitosa'; update tarjeta set puntos=nuevos_puntos where tarjeta.id_cliente = id_cliente and estatus = "Habilitada" ; update cliente set nombre_cliente = nombreCompleto where cliente.id_cliente = id_cliente; commit; else set error = 'Error: Empleado no existe'; end if; END // delimiter ; </pre>
Demostración
<pre> set @error=''; call actualizarSaldoNombre (2, 1, "Gustavo", @error) ; select @error; set @error=''; call actualizarSaldoNombre (20490687, -1, "Carlos", @error) ; select @error; </pre>
Resultados
 <p>The screenshot shows a table with one column labeled '@error'. The value in the row is 'Error: Empleado no existe'.</p>

<div> <div>@error</div> <div>Error: Sueldo menor al limite</div> </div>
Explicación de resultados
<p>Este procedimiento hace lo mismo que el anterior, a diferencia que este también actualiza el nombre del cliente y que fue creado en la base de datos del proyecto 'maquina_expendedora'.</p> <p>Este procedimiento hace uso de manejadores, y del disparador que anteriormente habíamos creado para evitar saldos negativos. Primeramente, se tiene como parámetro de entrada el identificador del cliente y saldo en puntos del cliente, y como parámetro de salida, el mensaje de error. Primero se verifica si el cliente existe mediante la consulta con count(), si existe se procede a hacer la actualización, si no, nos arroja el error que el cliente no existe. En caso de existir, nos actualiza el salario y el nombre del cliente, pero si este es saldo negativo, se activa el disparador que habíamos creado anteriormente y nos manda al manejador, este manejador establecerá el valor del mensaje de error a sueldo menor al límite, finalmente seguirá con la ejecución del procedimiento.</p> <p>En este caso, se hicieron dos pruebas, la primera fue actualizar el sueldo de cliente con uno que no existe, es por eso que nos arrojó el error de no existencia del empleado, y en el otro caso, se actualizó con una cantidad de saldo negativa, la cual resultó fallida igualmente, porque no podemos ingresar saldo menor a cero.</p>

Procedimiento
<pre> DROP FUNCTION IF EXISTS f_actualizar_saldo; DELIMITER // CREATE function f_actualizar_saldo (id_cliente int, nuevos_puntos int) RETURNS VARCHAR(100) DETERMINISTIC BEGIN DECLARE MENSAJE varchar(100); DECLARE CANT INT; DECLARE CONTINUE HANDLER FOR 1644 SET MENSAJE = 'Error: Sueldo menor al limite'; select count(*) into CANT from tarjeta where tarjeta.id_cliente = id_cliente and estatus = "Habilitada"; if CANT=1 then set MENSAJE = 'Actualizacion Exitosa'; update tarjeta set puntos=nuevos_puntos where tarjeta.id_cliente = id_cliente and estatus = "Habilitada" ; else set MENSAJE = 'Error: Empleado no existe'; </pre>

```

end if;
RETURN MENSAJE;
END //
DELIMITER ;

```

Demostración

```

SET @error = "";
SET @error = (f_actualizar_saldo(2, 10000));
select @error;

```

Resultados

@error
Error: Sueldo menor al limite

Explicación de resultados

Esta función tiene como parámetros de entrada el identificador de cliente y los puntos de la tarjeta el cual se desea asignar, y tiene como retorno cualquier cadena de 100 caracteres. Primero se verifica si el cliente existe, si no existe se cambia el mensaje de error y se retorna y finaliza la función. Caso contrario, realizará la actualización del salario, si este saldo es menor a 0, entonces el disparador se activa y nos manda al manejador, el cual cambiará el mensaje de error por el error presente. Se hizo una prueba, donde es para un cliente que sí existe, pero el saldo en puntos asignado es negativo, por lo que nos manda el error que es menor al límite establecido.

V. Conclusión y recomendaciones.

Con base a lo anteriormente mencionado, se puede concluir que el manejo de transacciones dentro de nuestro código permite mayor flexibilidad, sobre todo en el caso de que se presente un error que llegue a interrumpir la ejecución de alguna de las instrucciones dentro de la transacción; existiendo solamente dos estados posibles de finalización, que salga exitosa la transacción, o que se aborte y que todo el progreso sea desechado. En sí, lo consideramos una buena práctica que implementar a futuro, sobre todo, por si se llegase a considerar que la aplicación tenga algún tipo de excepciones o similares, o también dependiendo de la escalabilidad de la misma.

Con respecto a las prácticas de transacciones, fueron claras y sencillas, y no hubo problemas al momento de su elaboración, sobre todo, permitió mejorar las

habilidades de manejo de consultas y transacciones, solamente es cuestión de seguir practicando porque en ocasiones se puede llegar a olvidar la sintaxis, y por ende, se más toma tiempo consultando diferentes referencias en internet.

En definitiva, recomendamos adoptar este tipo de prácticas para implementar transacciones a futuro, porque poseen una vital importancia, sobre todo en el contexto bancario, donde la mayoría de las operaciones básicas financieras realizadas están salvaguardadas por transacciones en SQL para mantener un seguimiento de las modificaciones y tener mayor precisión en los datos.

Asimismo, con base a lo anteriormente mencionado, se concluye que la realización de las prácticas mediante la implementación de procedimientos almacenados, disparadores y manejadores fue de gran importancia para reconocer el impacto que tienen al momento de desarrollar consultas SQL que nos permitan automatizar o mejorar la eficiencia en nuestro sistema. Además, no se presentaron problemas al momento de resolverlas, si acaso solamente hubo faltas de información que mediante búsquedas en internet se ejemplificaba mejor cómo implementar dichos conceptos.

VI. Referencias bibliográficas.

1. Dickens, A. (2017, 13 abril). Transactions in SQL. Tutorial Point. Recuperado 4 de diciembre de 2022, de <https://www.tutorialspoint.com/sql/sql-transactions.htm>
2. Senzel, D. (2018, 1 febrero). Transactions in SQL. GeeksOf Geeks. Recuperado 4 de diciembre de 2022, de <https://www.geeksforgeeks.org/sql-transactions/>
3. S, R. A. (2022, 28 octubre). Stored Procedure in SQL: Benefits And How to Create It. Simplilearn.com. <https://www.simplilearn.com/tutorials/sql-tutorial/stored-procedure-in-sql>
4. Diller, M. (2022, 30 noviembre). CREATE TRIGGER (Transact-SQL) - SQL Server. Microsoft Learn. Recuperado 11 de diciembre de 2022, de <https://learn.microsoft.com/en-us/sql/t-sql/statements/create-trigger-transact-sql?view=sql-server-ver16>

5. DB2 11 - Application programming and SQL - Handlers in an SQL procedure.
(s. f.).

<https://www.ibm.com/docs/fi/db2-for-zos/11?topic=procedure-handlers-in-sql>

6. MySQL :: MySQL 8.0 Reference Manual :: 13.6.7.2 DECLARE . . . HANDLER
Statement. (s. f.).

<https://dev.mysql.com/doc/refman/8.0/en/declare-handler.html>