UNIVERSIDAD
NACIONAL
DEL NORDESTE

Facultad de Ciencias Exactas y Naturales Agrimensura

Cátedra: Base de Datos I

Año: 2025

Informe

Proyecto Integrador de Bases de Datos I

Equipo N°15

Integrantes:

- Ana Fiorella, Arduino De Michielis
- Pablo, Mansilla
- Almada, Tomas Emanuel
- Dana Florencia, Ramirez Cocomarola

Año: 2025

Capítulo I: introducción

Caso de estudio

Para la realización de este trabajo hemos elegido el diseño e implementación de una base de datos para un sistema de gestión de gimnasios.

Definición o planteamiento del problema

Actualmente, el gimnasio maneja de manera manual y desorganizada la información relacionada con sus alumnos, membresías, entrenadores y planes de entrenamiento. Esto genera los siguientes problemas:

- Almacenamiento desorganizado de datos: La información de los alumnos, membresías, pagos y planes de entrenamiento se mantiene de forma desorganizada y con inconsistencias en el formato, dificultando su manejo, accesibilidad y afectando negativamente a su integridad y utilidad.
- Falta de normalización: Los datos personales de los alumnos (nombre, apellido, teléfono, DNI, fecha de nacimiento, correo) y la información de membresías no siguen una estructura unificada, facilitando la duplicidad y redundancia de datos.
- Dificultades en las relaciones entre entidades: La asociación entre alumnos, entrenadores, planes de entrenamiento y horarios carece de una estructura relacional definida, afectando la eficiencia en las consultas y actualizaciones.
- Limitaciones en el control de transacciones: El registro manual de pagos y la asignación de planes de entrenamiento no garantizan la atomicidad y consistencia de las operaciones.

Objetivo del Trabajo Práctico

Objetivo General

Diseñar e implementar una base de datos relacional que centralice y gestione toda la información del gimnasio, optimizando el registro de socios, el control de membresías, la asignación de rutinas y la generación de reportes, con el fin de agilizar las operaciones y mejorar la eficiencia en la gestión.

Objetivos Específicos

- Identificar las entidades, atributos y relaciones necesarias para la gestión completa del gimnasio.
- Diseñar el modelo entidad-relación que represente la estructura de datos requerida por todos los actores del sistema (propietario, administrador, entrenadores).
- Normalizar la base de datos hasta la tercera forma normal (3FN) para eliminar redundancias y garantizar la integridad referencial.

- Implementar el esquema físico de la base de datos con las tablas, relaciones, restricciones y tipos de datos correspondientes.
- Desarrollar el diccionario de datos con la descripción de todas las tablas, campos, tipos de datos, restricciones y relaciones.
- Implementar procedimientos almacenados para automatizar operaciones críticas como:
    - Control de vencimiento de membresías
    - Registro automático de estados de pago
    - Validación de asignación de rutinas
- Diseñar e implementar índices estratégicos para optimizar el rendimiento de las consultas más frecuentes.
- Establecer los perfiles de usuario y permisos de acceso a nivel de base de datos según los roles identificados (dueño, administrador, entrenador).
- Validar el funcionamiento de la base de datos mediante consultas de prueba que demuestren la correcta gestión de todos los procesos del gimnasio.

Se espera que este trabajo pueda sentar las bases para el desarrollo futuro de una interfaz amigable para el usuario, priorizando eficiencia, seguridad y escalabilidad de la información gestionada.

Capítulo II: marco conceptual o referencial

Con este sistema, buscamos que el gimnasio sea más eficiente y automático. La idea es que manejar la información diaria sea mucho más fácil, y así transformarla en conocimiento útil para decisiones.

Impacto en la Gestión y el Análisis (El Eje Central es la Información):

- **Rendimiento y Estadísticas Detalladas:** Se generarán estadísticas clave sobre el rendimiento de instructores y usuarios. Esto permitirá identificar áreas de mejora, reconocer el desempeño superior y diseñar planes de servicio altamente efectivos.
- **Administración de Miembros y Personal (Gestión Integral de BD):** El sistema gestionará el ciclo de vida completo de los miembros (altas, bajas, historial) y la información laboral del personal. La precisión de estos datos es fundamental.
- **Análisis de Preferencias:** El enfoque estará en analizar cuáles son las rutinas o clases más solicitadas. Esto facilitará el lanzamiento de promociones y paquetes atractivos, basados en las elecciones reales de los clientes.

Beneficios Estratégicos y de Marketing:

- **Atracción y Retención:** La creación de ofertas personalizadas, basadas en el análisis de las preferencias de los usuarios, es fundamental. Esto no solo genera lealtad en los miembros existentes, sino que también funciona como un atractivo para nuevos

usuarios.
- **Mejora de la Imagen:** El uso de la información para optimizar los servicios incrementa la percepción general de calidad.

En resumen, el sistema busca ir más allá del registro de datos. Es una herramienta estratégica diseñada con el fin de transformar los datos en decisiones inteligentes, optimizar la experiencia del usuario y potenciar la rentabilidad y el prestigio del gimnasio.

Capítulo III: metodología seguida

El presente caso de estudio se llevó a cabo mediante la metodología de SCRUM.

- **Descripción de cómo se realizó este trabajo:**

El grupo se dividió las tareas en varias partes que se realizaron por separado, pero contando con el apoyo conjunto de cada miembro, en caso de necesitarla. Una vez cumplidos los plazos de tiempo, dichas partes se unifican y comprueba que funcione como se espere, bajo prioridades establecidas para dicho desarrollo.

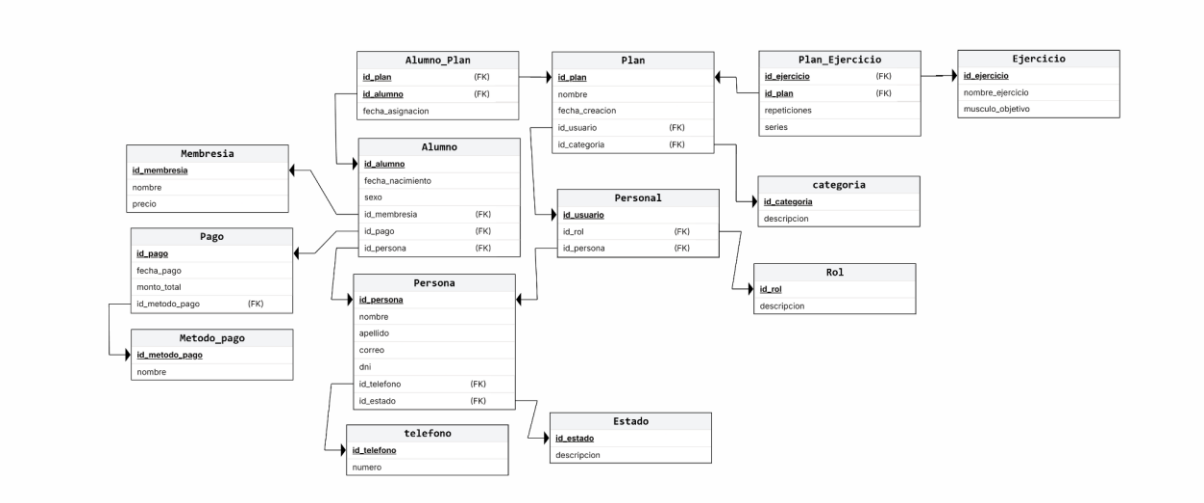
- **Herramientas**:

El grupo de trabajo se comunicó constantemente por WhatsApp, donde se asignaron las tareas correspondientes a cada miembro y se realizaron consultas para sus avances. Dichos avances luego fueron cargados a GitHub, donde se organizaron todos los archivos del trabajo.

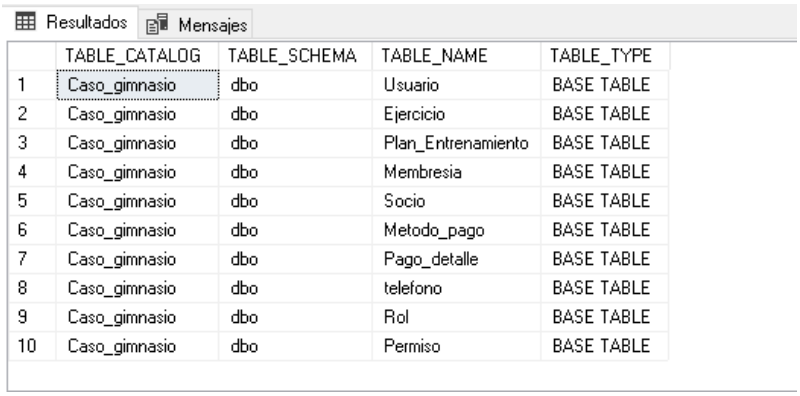
Capítulo IV: Desarrollo del tema y presentación de resultados

A continuación, se mostrará el diagrama relacional del caso trabajado.

Diagrama relacional



Diccionario de datos



Resultados | Mensajes

	TABLE_CATALOG	TABLE_SCHEMA	TABLE_NAME	TABLE_TYPE
1	Caso_gimnasio	dbo	Usuario	BASE TABLE
2	Caso_gimnasio	dbo	Ejercicio	BASE TABLE
3	Caso_gimnasio	dbo	Plan_Entrenamiento	BASE TABLE
4	Caso_gimnasio	dbo	Membresia	BASE TABLE
5	Caso_gimnasio	dbo	Socio	BASE TABLE
6	Caso_gimnasio	dbo	Metodo_pago	BASE TABLE
7	Caso_gimnasio	dbo	Pago_detalle	BASE TABLE
8	Caso_gimnasio	dbo	telefono	BASE TABLE
9	Caso_gimnasio	dbo	Rol	BASE TABLE
10	Caso_gimnasio	dbo	Permiso	BASE TABLE

Acceso al diccionario de datos:
https://github.com/TomasUNNE/ProyectoGrupo15/blob/main/doc/Diccionario%20de%20Datos.pdf

Tema: Manejo de Permisos a Nivel de usuarios de base datos

Este tema consiste en crear usuarios para la base de datos a los que puede otorgar distintos roles que definen las funciones a las que tienen acceso o que pueden usar para navegar o trabajar en la base de datos.

SQL Management Studio, por ejemplo, ya comienza con un “super administrador” el cual tiene acceso a todas las funciones del programa y desde esta misma es posible crear, organizar y controlar los permisos de los futuros usuarios.

Esto es importante para mantener un orden en la base de datos de una empresa, por ejemplo, limitar quienes tienen acceso a que funciones y evitar accidentes o modificaciones no esperadas que podrían llevar a un serio problema de organización y un gasto de tiempo que pudo haberse evitado. Por ejemplo:

```
--Se crean los usuarios a nivel servidor, sin roles asignados.

create login manuel with password='Password123';
create login juan with password='Password123';



--Se crea los usuarios con los login anteriores para una base de datos ESPECIFICA

USE Caso_gimnasio; ----- <-Verificar en que base de datos crear la instancia de estos usuarios.
GO

CREATE USER manuel FOR LOGIN manuel;
CREATE USER juan FOR LOGIN juan;
```

```sql
--- Se asignan distintos permisos a los usuarios creados para esta base de datos (caso_gimnasio)

-- Por ejemplo, manuel solo puede leer las tablas, es decir esta restringido a usar solo la sentencia SELECT
EXEC sp_addrolemember 'db_datareader', 'manuel';
go


--Probamos si los permisos funcionan.
EXECUTE AS USER = 'manuel'; --EXECUTE permite presentarse como el usuario mencionado, para ello, asegurarse de estar en una estancia de base de datos.

SELECT * FROM persona --Debe ver las tablas.

INSERT INTO Rol (id_rol, descripcion) VALUES (6, 'Seguridad');  ---No deberia poder insertar.

REVERT  ---Regresas al estado inicial, sin el user manuel activo.



--- Para el usuario Juan, se le otorga permisos de escritura, es decir, puedo usar unicamente sentencias INSERT
EXEC sp_addrolemember 'db_datawriter', 'juan';


--- Probamos...
EXECUTE AS USER = 'juan';
SELECT * FROM persona --No deberia poder ver las tablas.
INSERT INTO Rol (id_rol, descripcion) VALUES (6, 'Seguridad'); ---Deberia poder insertar.

REVERT --- Recuerda salir del usuario activo.
```

Otros tipos de roles que existen (Conocidos como roles fijos):

- db_owner: Los miembros del rol fijo de base de datos db_owner pueden realizar todas las actividades de configuración y mantenimiento en la base de datos, y también pueden borrar (drop) la base de datos en SQL Server. (En SQL Database y Azure Synapse, algunas actividades de mantenimiento requieren permisos a nivel de servidor y no se pueden realizar por db_owners).

- db_securityadmin: Los miembros del rol fijo de base de datos db_securityadmin pueden modificar la pertenencia a roles únicamente para roles personalizados y administrar permisos. Los miembros de este rol pueden elevar potencialmente sus privilegios y se deben supervisar sus acciones.

- db_accessadmin: Los miembros del rol fijo de base de datos db_accessadmin pueden agregar o eliminar el acceso a la base de datos para inicios de sesión de Windows, grupos de Windows e inicios de sesión de SQL Server.

- db_backupoperator: Los miembros del rol fijo de base de datos db_backupoperator pueden crear copias de seguridad de la base de datos.

- db_ddladmin: Los miembros del rol fijo de base de datos db_ddladmin pueden ejecutar cualquier comando del lenguaje de definición de datos (DDL) en una base de datos. Los miembros de este rol pueden potencialmente aumentar sus privilegios manipulando código que puede ser ejecutado bajo altos privilegios y sus acciones se deben supervisar.

- db_datawriter: Los miembros del rol fijo de base de datos db_datawriter pueden agregar, eliminar o cambiar datos en todas las tablas de usuario. En la mayoría de los casos de uso, este rol se combinará con la membresía db_datareader para permitir la lectura de los datos que se van a modificar.

- db_datareader: Los miembros del rol fijo de base de datos db_datareader pueden leer todos los datos de todas las tablas y vistas de usuario. Los objetos de usuario pueden existir en cualquier esquema, excepto sys e INFORMATION_SCHEMA.

- db_denydatawriter: Los miembros del rol fijo de base de datos db_denydatawriter no pueden agregar, modificar ni eliminar datos de tablas de usuario de una base de datos.

- db_denydatareader: Los miembros del rol fijo de base de datos db_denydatareader no pueden leer datos de las tablas y vistas de usuario dentro de una base de datos.

Luego tenemos un par de roles más especiales:

Estos roles y permisos se aplican principalmente a la base de datos "master" y suelen estar relacionados con la administración y configuración del servidor.

- dbmanager: Puede crear y eliminar bases de datos, convirtiéndose en el propietario. Tiene todos los permisos en las bases de datos que crea, pero no en otras.
- db_exporter: Aplicable a grupos de SQL dedicados de Azure Synapse Analytics. Permite actividades de exportación de datos con permisos como CREATE TABLE, ALTER ANY SCHEMA, ALTER ANY EXTERNAL DATA SOURCE, y ALTER ANY EXTERNAL FILE FORMAT.
- loginmanager: Puede crear y eliminar inicios de sesión en la base de datos "master" virtual.

Tema: Optimización de consultas a través de índices

Este tema fusiona los fundamentos teóricos de la indexación en SQL Server con los resultados empíricos obtenidos del estudio del caso práctico Caso_gimnasio, demostrando cómo la teoría se aplica directamente para resolver problemas de rendimiento en bases de datos a gran escala.

1. Marco Teórico: Fundamentos de Índices

1.1. Optimización de Consultas a través de Índices

SQL Server considera a los índices como estructuras de datos en disco utilizadas con el fin de reducir el tiempo y mejorar la eficiencia de las consultas realizadas a una tabla determinada. Estos índices contienen copias de los datos de la tabla, organizados con una estructura B-árbol que permite encontrar las filas asociadas a los valores de clave rápida y eficientemente.

Al ejecutar una consulta, el optimizador de consultas se encarga de evaluar cada método disponible para recuperar los datos. Luego, selecciona al más eficiente. Es decir, aquel que minimice el uso de recursos y tiempo.

1.2. Estructura de los Índices: El Árbol B

Ahondando en la estructura de los índices, un B-árbol es una estructura auto-balanceada capaz de organizar las copias de los datos de la tabla en diferentes nodos lógicos, facilitando y optimizando las operaciones de búsqueda.

Los nodos de un B-árbol se dividen en diferentes categorías o niveles:

- Nodo raíz: Nivel superior del árbol. Se considera como el punto de partida de cualquier búsqueda. Posee los rangos de claves y punteros a los nodos del siguiente nivel.
- Nodos intermedios: Aquellos que contienen rangos de claves y punteros a los nodos de nivel inmediatamente inferior.
- Nodos hoja: En ellos se almacena la información real de las claves del índice. Estos nodos se encuentran enlazados secuencialmente, permitiendo exploraciones eficientes.

1.3. Tipos de Índices

Los índices se pueden dividir en dos tipos principales:

Clustered (Agrupado): Determina el orden físico de almacenamiento de los datos en la tabla. Únicamente existe uno por tabla. Los nodos hoja de este tipo de índice contienen los datos completos de la tabla. (En nuestro caso, la PRIMARY KEY en id_persona crea este índice).

Nonclustered (No Agrupado): Es una estructura separada que contiene las claves seleccionadas y un puntero a la ubicación real de los datos. Es posible tener múltiples índices nonclustered por tabla. Sus nodos hoja contienen únicamente las claves del índice y punteros a la fila de datos.

2. Caso Práctico: Pruebas de Rendimiento en Caso_gimnasio

Para validar la teoría, se realizó un experimento sobre dos tablas idénticas que representan a la tabla Persona de Caso_Gimnasio (Persona_Sin_Indice y Persona_Con_Indice), cada una cargada con 6 millones de registros. El objetivo fue buscar personas por su correo electrónico.

2.1. Metodología

- Sin Índice: Se ejecutó una búsqueda de rango y una específica en Persona_Sin_Indice.
- Con Índice: Se creó un índice NONCLUSTERED en correo en la tabla Persona_Con_Indice y se repitieron las consultas.
- Métricas: Se midieron las Lecturas Lógicas (STATISTICS IO) y el Tiempo de CPU (STATISTICS TIME).

2.2. Resultados del escenario sin índice

Al buscar en Persona_Sin_Indice (que solamente tiene el índice Clustered en id_persona), el optimizador no encontró ningún "mapa" para la columna correo.

```
-- Consulta de Búsqueda Específica
PRINT '--- BUSQUEDA DE UN CORREO ESPECIFICO SIN INDICE EN CORREO ---';
SET STATISTICS TIME ON;
SET STATISTICS IO ON;

SELECT id_persona, nombre, apellido, correo
FROM Persona_Sin_Indice
WHERE correo = 'persona5820190@gimnasio.com'
ORDER BY correo ASC;

SET STATISTICS TIME OFF;
SET STATISTICS IO OFF;
GO

--El tiempo de CPU es de 1672ms y el de ejecución 1706ms.
--El total de lecturas realizadas fue de 53599
```

2.3. Resultados del escenario con índice

Se crearon índices Nonclustered en Persona_Con_Indice antes de la prueba.

```
-- ------------------------------------------------------------------------
-- 3) Definir índice y Medición con índice
-- ------------------------------------------------------------------------
-- 3.1 Crear el índice nonclustered
PRINT '--- 3.1) Creando Índice nonclustered en correo (IDX_Persona_Correo) ---';
CREATE NONCLUSTERED INDEX IDX_Persona_Correo ON Persona_Con_Indice(correo);
GO
```

```
-- 3.4 Búsqueda Específica con Índice
PRINT '--- 3.4) BUSQUEDA DE UN CORREO ESPECIFICO CON INDICE EN CORREO ---';
SET STATISTICS TIME ON;
SET STATISTICS IO ON;

SELECT id_persona, nombre, apellido, correo
FROM Persona_Con_Indice
WHERE correo = 'persona5820190@gimnasio.com'
ORDER BY correo ASC;

SET STATISTICS TIME OFF;
SET STATISTICS IO OFF;
GO

--La cantidad de lecturas lógicas realizadas disminuyó de manera drástica, a un total de 4 lecturas lógicas.
--El tiempo de CPU y de ejecución fue de 0ms, lo que significa que se ejecutó de forma casi instantanea.
```

3. Análisis

Los resultados del caso práctico son una demostración directa del marco teórico.

3.1 ¿Cuáles fueron los resultados en el escenario no indexado?

Como se indica en la teoría, el optimizador de consultas evaluó los métodos disponibles. Sin un índice en correo, el único método posible fue leer la tabla completa. Esto implicó recorrer

todos los nodos hoja del Índice Clustered (que contienen los datos de 6 millones de personas), resultando en 53,599 lecturas de página y un costo muy alto de CPU (1,672 ms).

3.2 ¿Cuáles fueron los resultados en el escenario indexado?

Al crear un índice NON CLUSTERED (IDX_Persona_Correo), se construyó una estructura B-Árbol secundaria específica para correo. Cuando el optimizador vio la misma consulta, seleccionó un plan de ejecución más eficiente: un Index Seek.

En lugar de leer 53,599 páginas, navegó el B-Árbol encontrando el valor exacto en solo 4 lecturas lógicas.

El costo fue tan bajo (0 ms) que la consulta fue prácticamente instantánea.

3.3 El Índice de Cobertura

Para la prueba de rango se usó un Índice de Cobertura (un índice Nonclustered con INCLUDE). Al incluir nombre, apellido y dni en el índice, el motor resolvió la consulta completa leyendo solo el índice, sin necesidad de tocar la tabla principal, reduciendo drásticamente las lecturas de 53,599 a solo 88 para un rango de 1000 registros.

```
-- Consulta de Búsqueda de Rango (La más costosa)
PRINT '--- 2) BUSQUEDA DE RANGOS SIN INDICE EN CORREO (Persona_Sin_Indice) ---';
SET STATISTICS TIME ON;
SET STATISTICS IO ON;

SELECT nombre, apellido, correo, dni
FROM Persona_Sin_Indice
WHERE correo BETWEEN 'persona100000@gimnasio.com' AND 'persona101000@gimnasio.com'
ORDER BY correo ASC;

SET STATISTICS TIME OFF;
SET STATISTICS IO OFF;
GO
--Table 'Worktable'. Scan count 0, logical reads 0, physical reads 0, page server reads 0,
--read-ahead reads 0, page server read-ahead reads 0, lob logical reads 0, lob physical reads 0,
--lob page server reads 0, lob read-ahead reads 0, lob page server read-ahead reads 0.

--Table 'Persona_Sin_Indice'. Scan count 1, logical reads 53599, physical reads 0, page server reads 0,
--read-ahead reads 53399, page server read-ahead reads 0, lob logical reads 0, lob physical reads 0,
--lob page server reads 0, lob read-ahead reads 0, lob page server read-ahead reads 0.

-- SQL Server Execution Times:
-- CPU time = 2734 ms,  elapsed time = 3177 ms.

--Las lecturas lógicas fueron 53599
--El tiempo de la CPU fue de 2734ms, lo cual es bastante ineficiente.
--El tiempo de ejecución fue de 3177ms.
```

```
-- 3.2 Crear un Índice de COBERTURA
PRINT '--- 3.2) Creando Índice de Cobertura (IDX_Persona_Correo_Covering) ---';
CREATE NONCLUSTERED INDEX IDX_Persona_Correo_Covering
ON Persona_Con_Indice(correo)
INCLUDE (nombre, apellido, dni);
GO
```

```sql
-- 3.3 Prueba de Rango con Índice de Cobertura
PRINT '--- 3.3) PRUEBA DE RANGO CON ÍNDICE DE COBERTURA ---';
SET STATISTICS TIME ON;
SET STATISTICS IO ON;

SELECT nombre, apellido, correo, dni
FROM Persona_Con_Indice
WHERE correo BETWEEN 'persona100000@gimnasio.com' AND 'persona101000@gimnasio.com'
ORDER BY correo ASC;

SET STATISTICS IO OFF;
SET STATISTICS TIME OFF;
GO

--Resultados obtenidos
--Table 'Persona_Con_Indice'. Scan count 1, logical reads 88, physical reads 0, page server reads 0, read-ahead reads 0,
--page server read-ahead reads 0, lob logical reads 0, lob physical reads 0, lob page server reads 0, lob read-ahead reads 0,
--lob page server read-ahead reads 0.

 --SQL Server Execution Times:
 --CPU time = 15 ms,  elapsed time = 84 ms.

--La cantidad de lecturas lógicas realizadas disminuyó de manera drástica.
--El tiempo de CPU fue de 15ms y de ejecución fue de 84ms, lo que significa que se ejecutó muy rapidamente.
```

4. Conclusión General

Este estudio ilustra la importancia de la estrategia de indexación. Si bien un tiempo de respuesta de 1672 ms sin índice puede parecer irrelevante para el usuario, el impacto a nivel de sistema es catastrófico. La consulta sin índice consumió un aproximado de 13,400 veces más recursos (53,599 lecturas lógicas) de lo necesario, en comparación con el sistema optimizado, que resolvió la misma tarea casi instantáneamente (0ms y 4 lecturas).

En síntesis, es indispensable utilizar una buena estrategia de indexación si se tiene la intención de mantener un alto rendimiento y una experiencia de usuario óptima.

Tema: procedimientos y funciones almacenadas

1. Resumen General del Tema:

Al utilizar Procedimientos Almacenados Y Funciones dentro de la base de datos SQL, estos objetos permiten encapsular lógica de negocio (conjunto de reglas, políticas, y procesos que definen cómo opera y gestiona los datos una organización), operaciones complejas y reaccionar automáticamente a eventos en las tablas, mejorando la eficiencia, seguridad y mantenibilidad.

2. Procedimiento almacenado:

Es un objeto que se crea con la sentencia CREATE PROCEDURE y se invoca con la sentencia CALL. Un procedimiento puede tener cero o muchos parámetros de entrada y cero o muchos parámetros de salida. Se pueden utilizar para validar datos, controlar el acceso o reducir el tráfico de red. Los procedimientos pueden realizar las siguientes operaciones lo que los asemejan a las construcciones de otros lenguajes de programación:

- Aceptar parámetros de entrada y devolver varios valores en forma de parámetros de salida al programa que realiza la llamada.

- Contener instrucciones de programación que realicen operaciones en la base de datos. Entre otras, pueden contener llamadas a otros procedimientos.
- Devolver un valor de estado a un programa que realiza una llamada para indicar si la operación se ha realizado correctamente o se han producido errores, y el motivo de estos.

3. Función almacenada:

Es un objeto que se crea con la sentencia CREATE FUNCTION y se invoca con la sentencia SELECT o dentro de una expresión. Una función puede tener cero o muchos parámetros de entrada y siempre devuelve un valor, asociado al nombre de la función.

Diferencias entre Procedimientos y Funciones Almacenadas (MySQL vs. Otros Motores)

La principal distinción entre los procedimientos almacenados (Stored Procedures - SPs) y las funciones almacenadas (Stored Functions - SFs) radica en su propósito y en cómo manejan los datos de entrada y salida.

En MySQL (CREATE PROCEDURE vs. CREATE FUNCTION)

Característica	Procedimiento Almacenado (PROCEDURE)	Función Almacenada (FUNCTION)
Invocación	Se invoca con la sentencia CALL	Se invoca con la sentencia SELECT o dentro de una expresión
Valor de Retorno	Puede devolver varios valores en forma de parámetros de salida (OUT/INOUT)	Siempre devuelve un único valor, asociado al nombre de la función, y requiere la cláusula RETURNS
Tipos de Parámetros	Acepta parámetros de entrada (IN), salida (OUT), y entrada/salida (INOUT)	Todos los parámetros son de entrada (IN). No es válido especificar OUT o INOUT
Resultados (Conjunto de Filas)	Puede contener sentencias SELECT que devuelvan un conjunto de resultados (result set)	No puede devolver un conjunto de resultados. Si un SELECT no tiene una cláusula INTO, se producirá un error.
Transacciones	Puede contener sentencias de transacción SQL explícitas o implícitas, como COMMIT o ROLLBACK	No puede contener sentencias que realicen commit o rollback explícito o implícito.
Manipulación de Datos	Típicamente utilizada para modificar datos (Modifies SQL Data)	Para poder crearse, debe ser declarada como DETERMINISTIC, NO SQL, o READS SQL DATA (aunque MODIFIES SQL DATA también es una característica posible).

4. Respecto a Otros Motores (ej. SQL Server)

En motores como SQL Server o Azure SQL Database, un procedimiento almacenado es un grupo de (1) instrucciones Transact-SQL o una referencia a un (2) método CLR. Mientras que las diferencias conceptuales (procedimientos para acciones con efectos secundarios y funciones para cálculos que devuelven un valor) se mantienen, las diferencias prácticas residen en la sintaxis, el lenguaje de programación (T-SQL o CLR en SQL Server vs. SQL/Control Structures en MySQL), y las características específicas de manejo de tipos y errores: • Lenguaje/Entorno: Los (3) SPs de SQL Server pueden usar métodos de Common Runtime Language (CLR) de Microsoft .NET Framework. MySQL soporta rutinas escritas solo en SQL, e ignora la característica LANGUAGE si se especifica otra cosa. • Parámetros de Salida: Ambos permiten devolver valores a través de parámetros de salida. • Valor de Estado:

Los procedimientos de SQL Server pueden devolver un valor de estado al programa que realiza la llamada para indicar éxito o errores. En MySQL, esto se gestiona típicamente mediante parámetros OUT o manejadores de errores.

5. Ventajas de Usar Los Procedimientos

La utilización y buena aplicación de los procedimientos en una base de datos SQL trae grandes ventajas como:

1. Seguridad Reforzada: Los procedimientos almacenados actúan como guardianes de los datos subyacentes, lo cual simplifica y fortalece los niveles de seguridad.

    1.1. Protección de Objetos: Varios usuarios y programas cliente pueden realizar operaciones en los objetos de la base de datos subyacentes a través de un procedimiento, aunque no tengan permisos directos sobre esos objetos. El procedimiento controla qué actividades se llevan a cabo y protege las tablas.

    1.2. Permisos Simplificados: Esto elimina la necesidad de conceder permisos en cada nivel de objetos. Por ejemplo, acciones como TRUNCATE TABLE no tienen permisos que se puedan conceder directamente al usuario, pero se pueden ampliar los permisos para truncar la tabla al usuario al que se conceda el permiso EXECUTE para el módulo que contiene la instrucción.

    1.3. Prevención de Inyección SQL: El uso de parámetros en los procedimientos ayuda a protegerse contra ataques por inyección de código SQL. Dado que la entrada de parámetros se trata como un valor literal y no como código ejecutable, es más difícil para un atacante insertar comandos maliciosos en las instrucciones.

    1.4. Ocultación de la Arquitectura: Cuando una aplicación llama a un procedimiento a través de la red, solo la llamada es visible. Esto evita que los usuarios malintencionados vean los nombres de los objetos, las tablas de la base de datos o busquen datos críticos.

    1.5. Contexto de Seguridad: Se puede utilizar la cláusula EXECUTE AS (o SQL SECURITY en MySQL) para habilitar la suplantación de otro usuario o permitir que las aplicaciones realicen actividades sin necesidad de contar con permisos directos sobre los objetos subyacentes.

2. Tráfico de Red Reducido

    - El código de un procedimiento se ejecuta en un único lote de código. Esto reduce significativamente el tráfico de red entre el servidor y el cliente. Únicamente se envía a través de la red la llamada para ejecutar el procedimiento, mientras que, sin esta encapsulación, cada línea de código tendría que enviarse por separado.

3. Rendimiento Mejorado

    - De forma predeterminada, un procedimiento se compila la primera vez que se ejecuta, y el plan de ejecución creado se reutiliza en posteriores ejecuciones.

    - Debido a que el procesador de consultas no tiene que crear un plan nuevo cada vez, normalmente necesita menos tiempo para procesar el procedimiento.

4. Reutilización del Código: Cualquier operación de base de datos redundante es un candidato perfecto para la encapsulación en un procedimiento.

    - Esto elimina la necesidad de escribir el mismo código varias veces, reduciendo las inconsistencias de código.

    - Permite que cualquier usuario o aplicación con los permisos necesarios pueda acceder y ejecutar el código centralizado.

5. Mantenimiento Más Sencillo: Al llamar las aplicaciones cliente a procedimientos y mantener las operaciones de base de datos en la capa de datos, solo se deben actualizar los cambios de los procesos en la base de datos subyacente.

    - El nivel de aplicación permanece independiente y no necesita tener conocimiento sobre los cambios realizados en los diseños, las relaciones o los procesos internos de la base de datos.

5.1. Aplicación de los temas en nuestro modelo de datos

Para nuestro modelo de base de datos Caso_gimnasio, la implementación de rutinas almacenadas puede ayudar al potenciamiento de la integridad de los datos y la eficiencia operativa. Los SPs al ser bloques de procesos complejos simplifican los procesos de múltiples pasos, transacciones y modificaciones de datos. Casos de uso de Procedimientos Almacenados:

Manejo de Pagos Mensuales: Un SP podría gestionar la renovación de la Membresía, actualizando el estado del Alumno y registrando un nuevo Pago. Ejemplo: Scrip_Renovacion_Membresia_SPs.sql

```
-- Actualiza el pago de un alumno para renovar su membresía mediante un
Proceso Almacenado

USE Caso_gimnasio;



CREATE PROCEDURE Pa_RenovarMembresia

   @id_alumno INT,

   @id_metodo_pago INT

AS

BEGIN

   DECLARE @precio FLOAT;

   DECLARE @nuevo_id INT;
```

```
    -- Obtener precio de la membresía
    SELECT @precio = m.precio
    FROM Alumno a
    JOIN Membresia m ON a.id_membresia = m.id_membresia
    WHERE a.id_alumno = @id_alumno;

    -- Generar nuevo ID de pago
    SELECT @nuevo_id = ISNULL(MAX(id_pago), 0) + 1 FROM Pago;

    -- Registrar pago
    INSERT INTO Pago (id_pago, fecha_pago, monto_total, id_metodo_pago)
    VALUES (@nuevo_id, GETDATE(), @precio, @id_metodo_pago);

    -- Actualizar alumno con nuevo pago
    UPDATE Alumno SET id_pago = @nuevo_id
    WHERE id_alumno = @id_alumno;
END;
```

Casos de uso de Funciones Almacenadas (3) (SFs) Las SFs son perfectas para realizar cálculos reutilizables que se pueden integrar directamente en consultas SELECT o cláusulas WHERE:

Consultas de Proximidad a Vencimientos Uns SFs que devuelva una tabla con los datos de los alumnos y los días faltantes para el vencimiento de su membresía. Ejemplo: Scrip_Dias_Prox_Vencer_SFs.sql

```
--Este Script Muestra las membresías de los alumnos que están por vencer
en los próximos 30 días
```

```
--lo procesa añadiendo un mes a la fecha de pago y comparándola con la
fecha actual



USE Caso_gimnasio;



CREATE FUNCTION fn_MembresiasProximasVencer()

RETURNS TABLE

AS

RETURN

   SELECT a.id_alumno, p.nombre, p.apellido,

          DATEDIFF(DAY, GETDATE(), DATEADD(MONTH, 1, pg.fecha_pago)) AS
dias_para_vencer

   FROM Alumno a

   JOIN Persona p ON a.id_persona = p.id_persona

   JOIN Pago pg ON a.id_pago = pg.id_pago

   WHERE  DATEDIFF(DAY,  GETDATE(),  DATEADD(MONTH,  1,  pg.fecha_pago))
BETWEEN 0 AND 30

   AND p.id_estado = 1;
```

5.2. Otras Ventajas

Agregando Lógica Condicional dentro de un Procedimiento Almacenado: Para implementar lógica de acción "en función de" (es decir, ejecutar diferentes comandos basados en condiciones), MySQL proporciona estructuras de control en el cuerpo del procedimiento o función. Esto permite que se puedan utilizar:

1. Instrucciones Condicionales (IF-THEN-ELSE/CASE): Permiten ejecutar un bloque de sentencias u otro según si una condición es verdadera. ◦ Ejemplo de uso: En un SP de registro de pago, podría usar IF para verificar el monto (monto_total en Pago) y, si es inferior al precio de la Membresía, registrar el alumno en un estado "Pendiente" en la tabla Estado.

2. Instrucciones Repetitivas o Bucles (LOOP, REPEAT, WHILE): Permiten iterar sobre bloques de código. Son esenciales cuando se trabaja con cursores, que son estructuras que permiten recorrer secuencialmente un conjunto de filas. ◦ Ejemplo de

uso: Un SP podría usar un cursor para recorrer todos los alumnos con membresía vencida, y dentro de un bucle WHILE, actualizar el id_estado del alumno en la tabla Persona. Al trabajar con cursores, se debe manejar el error NOT FOUND (SQLSTATE '02000') con un HANDLER para saber cuándo salir del bucle.

Potenciando el Script en la Seguridad: El uso de procedimientos almacenados es una de las maneras más efectivas de potenciar la seguridad de su script y su modelo de datos.

Seguridad Reforzada mediante SPs:

1. Control de Permisos Granular: El Script puede permitir que varios usuarios o programas realicen operaciones sobre los objetos subyacentes (tablas como Persona, Pago, Personal) a través de un procedimiento, sin que esos usuarios o programas tengan permisos directos sobre los objetos. El procedimiento actúa como un guardián. Esto simplifica drásticamente los niveles de seguridad.

2. Protección contra Inyección de Código SQL: Al utilizar parámetros en sus procedimientos (como IN, OUT, INOUT), la entrada se trata como un valor literal y no como código ejecutable. Esto hace mucho más difícil para un atacante insertar comandos maliciosos en las instrucciones Transact-SQL del procedimiento y poner en peligro la seguridad.

3. Ocultación de la Arquitectura de la Base de Datos: Cuando una aplicación cliente llama a un SP a través de la red, solo la llamada al procedimiento es visible. Esto impide que usuarios malintencionados vean los nombres de los objetos, las tablas o las instrucciones Transact-SQL internas, dificultando la búsqueda de datos críticos.

4. Contexto de Seguridad (SQL SECURITY): En MySQL, se puede definir el contexto de seguridad con el que se ejecuta la rutina.

    - DEFINER (Predeterminado): La rutina se ejecuta con los privilegios de la cuenta que la creó (el DEFINER). Esto permite que usuarios con pocos privilegios ejecuten operaciones complejas (como insertar en tablas sensibles), siempre y cuando el definidor tenga esos permisos.

    - INVOKER: La rutina se ejecuta con los privilegios del usuario que la está invocando.

6. Resumen General del Tema:

Al utilizar Procedimientos Almacenados Y Funciones dentro de la base de datos SQL, estos objetos permiten encapsular lógica de negocio (conjunto de reglas, políticas, y procesos que definen cómo opera y gestiona los datos una organización), operaciones complejas y reaccionar automáticamente a eventos en las tablas, mejorando la eficiencia, seguridad y mantenibilidad.

En resumen, integrar procedimientos y funciones almacenadas en nuestro script no solo aumenta la eficiencia mediante el código reutilizable y la compilación, sino que crea una capa de seguridad esencial al aislar la lógica de negocio y proteger los objetos de la base de datos subyacentes contra accesos directos e inyecciones SQL.

Tema: Transacciones y Transacciones Anidadas

Manejo de Transacciones y Transacciones Anidadas en la Base de Datos "*Caso_gimnasio*"

El modelo de datos del *Caso_gimansio* representa un sistema de gestión integral que está compuesta por entidades como *Persona, Alumno, Membresía, Pago, Plan y Personal*, entre otras. En este contexto, el manejo correcto de las transacciones resulta fundamental para garantizar la integridad, coherencia y confiabilidad de la información almacenada, especialmente ante escenarios donde se realizan operaciones simultáneas o dependientes entre múltiples tablas.

1. Conceptos y Propiedades ACID

Una transacción en una base de datos representa una unidad lógica de trabajo compuesta por una o varias operaciones SQL que deben ejecutarse en su totalidad o no ejecutarse en absoluto.

Las propiedades ACID son esenciales en este contexto:

- **Atomicidad:** garantiza que todas las operaciones se ejecuten como un bloque indivisible. Si al registrar un pago en la tabla *Pago* ocurre un fallo, el sistema realiza un ROLLBACK, eliminando los cambios parciales.

- **Consistencia:** asegura que los datos cumplan con las reglas y restricciones de integridad (como claves foráneas entre *Alumno* y *Membresía*, o *Pago* y *Metodo_pago*).

- **Aislamiento:** permite que múltiples usuarios realicen operaciones simultáneas sin interferir entre sí, evitando que transacciones incompletas sean visibles para otros procesos.

- **Durabilidad:** garantiza que, una vez confirmado un COMMIT, los cambios queden almacenados permanentemente, incluso frente a fallos del sistema.

2. Aplicación Práctica en el Modelo del Gimnasio

En nuestra base de datos *Caso_gimnasio,* el uso de transacciones es esencial en operaciones que afectan múltiples entidades relacionadas. El alta de un nuevo alumno es el ejemplo perfecto, ya que requiere insertar datos en *teléfono, Persona, Pago* y *Alumno* de manera coordinada.

Usaremos la sintaxis moderna de T-SQL con BEGIN TRY… CATCH para un manejo robusto de errores.

Inserción inicial obligatoria:

```
-- Inserción inicial necesaria (se trabaja sobre un lote de datos de 9999 aunque no es necesario).
INSERT INTO Metodo_pago (id_metodo_pago, nombre) VALUES (1, 'Efectivo');
INSERT INTO Membresia (id_membresia, nombre, precio) VALUES (1, 'Básica', 29.99);
```

Ejemplo 1: Transacción Exitosa (Atomicidad y Durabilidad)

Se inscribe un nuevo alumno. Las 4 inserciones (telefono, Persona, Pago, Alumno) deben tener éxito. Si todas lo logran, el COMMIT hace los cambios permanentes (Durabilidad).

```sql
--Transacción Exitosa
GO
PRINT '--- INICIANDO TRANSACCIÓN EXITOSA ---';
BEGIN TRAN inscribir_alumno;
BEGIN TRY;

    -- 1. Insertar Telefono
    INSERT INTO telefono (numero) VALUES ('555-1234');
    DECLARE @NuevoTelefonoID INT = SCOPE_IDENTITY();
    /*DECLARE: define variables en contextos como procedimientos almacenados,
    funciones o scripts por lotes, en este caso, scripts por lote
    SCOPE_IDENTITY(): devuelve el último valor de identidad insertado en una columna
    de identidad en el mismo ámbito.*/

    -- 2. Insertar Persona (Asumimos id_estado = 1 existe)
    INSERT INTO Persona (nombre, apellido, correo, dni, id_telefono, id_estado)
    VALUES ('Juan', 'Perez', 'juan.perez@email.com', 12345678, @NuevoTelefonoID, 1);
    DECLARE @NuevoPersonaID INT = SCOPE_IDENTITY();

    -- 3. Insertar Pago (id_pago es manual en tu esquema)
    DECLARE @NuevoPagoID INT = 10000;
    INSERT INTO Pago (id_pago, fecha_pago, monto_total, id_metodo_pago)
    VALUES (@NuevoPagoID, GETDATE(), 5000.0, 1);

    -- 4. Insertar Alumno
    INSERT INTO Alumno (fecha_nacimiento, sexo, id_membresia, id_persona, id_pago)
    VALUES ('1990-05-15', 'M', 1, @NuevoPersonaID, @NuevoPagoID);

    -- Si llegamos aquí sin errores, confirmamos la transacción
    COMMIT TRAN inscribir_alumno;
    PRINT '--- TRANSACCIÓN EXITOSA (COMMIT) ---';

END TRY
BEGIN CATCH
    -- Si algo falla, revertimos TODO
    ROLLBACK TRAN inscribir_alumno;
    PRINT '--- TRANSACCIÓN FALLIDA (ROLLBACK) ---';
    PRINT 'Error: ' + ERROR_MESSAGE();
END CATCH;
GO
```

Resultado de la ejecución del script:

```
--- INICIANDO TRANSACCIÓN EXITOSA ---

(1 fila afectada)

(1 fila afectada)

(1 fila afectada)

(1 fila afectada)
--- TRANSACCIÓN EXITOSA (COMMIT) ---

Hora de finalización: 2025-11-15T22:09:22.6757499-03:00
```

Ejemplo 2: Transacción Fallida (Atomicidad y Consistencia)

Intentamos inscribir a “Ana Gómez” usando el mismo DNI (12345678) de Juan Pérez. La transacción insertará el teléfono, pero fallará en Persona debido a la restricción UQ_DNI

(Consistencia). El ROLLBACK se activará, revirtiendo todas las operaciones, incluyendo la inserción del teléfono. Esto demuestra la Atomicidad: o todo o nada.

```
--Transacción Fallida
GO
PRINT '--- INICIANDO TRANSACCIÓN FALLIDA ---';
BEGIN TRAN inscribir_alumno_fallo;
BEGIN TRY;

    -- 1. Insertar Telefono (Se ejecuta)
    INSERT INTO telefono (numero) VALUES ('555-9876');
    DECLARE @NuevoTelefonoID INT = SCOPE_IDENTITY();
    PRINT 'Teléfono creado con ID: ' + STR(@NuevoTelefonoID);

    -- 2. Insertar Persona (Aquí sucedera la violación de restricción)
    INSERT INTO Persona (nombre, apellido, correo, dni, id_telefono, id_estado)
    VALUES ('Ana', 'Gomez', 'ana.gomez@email.com', 12345678, @NuevoTelefonoID, 1); -- DNI REPETIDO

    -- (El código de abajo NUNCA se ejecuta)
    DECLARE @NuevoPersonaID INT = SCOPE_IDENTITY();
    DECLARE @NuevoPagoID INT = 10001;

    INSERT INTO Pago (id_pago, fecha_pago, monto_total, id_metodo_pago)
    VALUES (@NuevoPagoID, GETDATE(), 5000.0, 1);

    INSERT INTO Alumno (fecha_nacimiento, sexo, id_membresia, id_persona, id_pago)
    VALUES ('1995-10-20', 'F', 1, @NuevoPersonaID, @NuevoPagoID);

    COMMIT TRAN inscribir_alumno_fallo;

END TRY
BEGIN CATCH
    -- El CATCH se activa por la violación de DNI único
    ROLLBACK TRAN inscribir_alumno_fallo;
    PRINT '--- TRANSACCIÓN FALLIDA (ROLLBACK) ---';
    PRINT 'Error: ' + ERROR_MESSAGE();

    -- Verificación: El teléfono '555-9876' NO existirá en la BD
    SELECT COUNT(*) AS 'telefono_ana_gomez_existe' FROM telefono WHERE numero = '555-9876'; -- (Devolverá 0)
END CATCH;
GO
```

Resultado de la ejecución del script:

```
--- INICIANDO TRANSACCIÓN FALLIDA ---

(1 fila afectada)
Teléfono creado con ID:      10007

(0 filas afectadas)
--- TRANSACCIÓN FALLIDA (ROLLBACK) ---
Error: Violation of UNIQUE KEY constraint 'UQ_DNI'. Cannot insert duplicate key in object 'dbo.Persona'. The duplicate key value is (12345678).

(1 fila afectada)

Hora de finalización: 2025-11-15T22:20:31.4657908-03:00
```

	telefono_ana_gomez_existe
1	0

3. Transacciones Anidadas en SQL Server

El modelo de transacciones anidadas es útil cuando las operaciones se dividen en subprocesos. Sin embargo, SQL Server tiene un manejo particular de esto que es crucial entender.

3.1 Transacciones Anidadas (@@TRANCOUNT)

En T-SQL, BEGIN TRAN incrementa un contador (@@TRANCOUNT). COMMIT TRAN lo decrementa. La transacción solo se confirma realmente cuando @@TRANCOUNT vuelve a 0. El problema es que un ROLLBACK TRAN (sin importar qué tan “anidado” esté) siempre revierte toda la transacción (vuelve @@TRANCOUNT a 0) y deshace todo el trabajo.

Ejemplo:

```
--Transacción Anidada
GO
PRINT '--- DEMOSTRACIÓN DE TRANSACCION ANIDADA ---';
PRINT '@@TRANCOUNT inicial: ' + STR(@@TRANCOUNT); -- Debe ser 0

BEGIN TRAN transaccion_1; -- transaccion_1 (Externa)
    PRINT 'BEGIN transaccion_1. @@TRANCOUNT: ' + STR(@@TRANCOUNT); -- Debe ser 1
    INSERT INTO Pago (id_pago, fecha_pago, monto_total, id_metodo_pago)
    VALUES (10001, GETDATE(), 100.0, 1);

    BEGIN TRAN transaccion_2; -- transaccion_2 (Anidada/Interna)
        PRINT '  BEGIN transaccion_2. @@TRANCOUNT: ' + STR(@@TRANCOUNT); -- Debe ser 2
        INSERT INTO Pago (id_pago, fecha_pago, monto_total, id_metodo_pago)
        VALUES (10002, GETDATE(), 200.0, 1);

        PRINT '  ROLLBACK GENERAL...';
        ROLLBACK;
        PRINT '  ROLLBACK ejecutado. @@TRANCOUNT: ' + STR(@@TRANCOUNT); -- Debe ser 0

-- Como el ROLLBACK ya bajó el contador a 0, este IF evitará errores
IF @@TRANCOUNT > 0
    COMMIT TRAN transaccion_1;
ELSE
    PRINT 'No se puede hacer COMMIT transaccion_1, la transacción ya fue revertida completamente.';
GO

-- Verificación: no deberían existir
SELECT * FROM Pago WHERE id_pago IN (10001, 10002);
```

Resultado de la ejecución del script:

```
--- DEMOSTRACIÓN DE TRANSACCION ANIDADA ---
@@TRANCOUNT inicial:          0
BEGIN transaccion_1. @@TRANCOUNT:          1

(1 fila afectada)
  BEGIN transaccion_2. @@TRANCOUNT:          2

(1 fila afectada)
  ROLLBACK GENERAL...
  ROLLBACK ejecutado. @@TRANCOUNT:          0
No se puede hacer COMMIT transaccion_1, la transacción ya fue revertida completamente.

(0 filas afectadas)

Hora de finalización: 2025-11-15T22:34:54.0677069-03:00
```

	fecha_pago	monto_total	id_pago	id_metodo_pago

3.2 Puntos de Guardado (SAVE TRANSACTION)

Lo que la mayoría de la gente busca con “transacciones anidadas” es la capacidad de revertir parcialmente una transacción. Para esto se usa SAVE TRANSACTION.

Escenario: registramos un pago. Creamos un SAVEPOINT. Luego intentamos otra operación (ej. actualizar la membresía) y falla. Queremos revertir solo la actualización, pero mantener el pago registrado.

Ejemplo:

```
--Transacción Anidada con punto de guardado
GO
PRINT '--- USANDO SAVE TRANSACTION (PUNTOS DE GUARDADO) ---';
BEGIN TRAN transaccion_principal; -- @@TRANCOUNT = 1
PRINT 'BEGIN transaccion_principal. @@TRANCOUNT: ' + STR(@@TRANCOUNT);

-- Operación 1: Insertar el primer pago
INSERT INTO Pago (id_pago, fecha_pago, monto_total, id_metodo_pago) VALUES (10001, GETDATE(), 1000.0, 1);
PRINT 'Pago (10001) insertado.';

-- Creamos un punto de guardado
SAVE TRANSACTION punto_de_guardado_1;
PRINT 'SAVEPOINT creado (punto_de_guardado_1).';

-- Operación 2: Intentar insertar un pago duplicado (forzar error)
BEGIN TRY
    PRINT 'Intentando insertar Pago (10001) de nuevo...';
    INSERT INTO Pago (id_pago, fecha_pago, monto_total, id_metodo_pago) VALUES (10001, GETDATE(), 2000.0, 1);
END TRY
BEGIN CATCH
    PRINT 'Error detectado: ' + ERROR_MESSAGE();
    -- Revertimos SÓLO la Operación 2, volviendo al savepoint
    ROLLBACK TRANSACTION punto_de_guardado_1;
    PRINT 'ROLLBACK al SAVEPOINT ejecutado. La Operación 1 (Pago 10001) sigue en la transacción.';
END CATCH;

-- La transacción principal SIGUE ACTIVA (@@TRANCOUNT = 1)
-- Confirmamos la transacción (esto guarda el Pago 10001)
IF @@TRANCOUNT > 0
BEGIN
    COMMIT TRAN transaccion_principal;
    PRINT 'COMMIT transaccion_principal ejecutado.';
END
GO

-- Verificación: El pago 10001 SÍ debe existir.
SELECT * FROM Pago WHERE id_pago = 10001;
```

Resultado de la ejecución del script:

```
--- USANDO SAVE TRANSACTION (PUNTOS DE GUARDADO) ---
BEGIN transaccion_principal. @@TRANCOUNT:         1

(1 fila afectada)
Pago (10001) insertado.
SAVEPOINT creado (punto_de_guardado_1).
Intentando insertar Pago (10001) de nuevo...

(0 filas afectadas)
Error detectado: Violation of PRIMARY KEY constraint 'PK_Pago'. Cannot insert duplicate key in object 'dbo.Pago'. The duplicate key value is (10001).
ROLLBACK al SAVEPOINT ejecutado. La Operación 1 (Pago 10001) sigue en la transacción.
COMMIT transaccion_principal ejecutado.

(1 fila afectada)

Hora de finalización: 2025-11-15T22:45:04.4839272-03:00
```

	fecha_pago	monto_total	id_pago	id_metodo_pago
1	2025-11-15	1000	10001	1

Este manejo jerárquico usando SAVE TRANSACTION permite mayor control y recuperación ante errores parciales, logrando el objetivo de modularidad.

4. Conclusión del tema

En síntesis, el modelo *Caso_gimnasio* se beneficia ampliamente de la implementación de transacciones, al tratarse de un sistema con alta interdependencia entre entidades. Su uso garantiza la integridad de los datos (Atomicidad, Consistencia) y la fiabilidad del sistema (Durabilidad, Aislamiento).

En términos prácticos, las transacciones actúan como un mecanismo de control de calidad, y mientras T-SQL no soporta “transacciones anidadas” verdaderas con ROLLBACK independientes, sí provee el mecanismo SAVE TRANSACTION para lograr puntos de reversión parciales dentro de una transacción mayor.

5. Referencias Bibliográficas

- Elmasri, R., y Navathe, S. B. (2007). *Fundamentos de sistemas de bases de datos* (5ª ed.). Madrid: Pearson Educación.
- Ramakrishnan, R., y Gehrke, J. (2003). *Database Management Systems* (3ª ed.). New York: McGraw-Hill.
- Silberschatz, A., Korth, H. F., y Sudarshan, S. (2011). *Database System Concepts* (6ª ed.). New York: McGraw-Hill.