



Universidad Nacional del Nordeste



Facultad de Ciencias Exactas y Naturales y Agrimensura

Base de Datos 1

Año: 2025

Alumnos:	D.N.I. N°:
Bazzola Gabriel Esteban	45374342
Mazzanti Marcos Santino	45760887
Rojas Yaccuzzi Joaquin	45760608
Zambrano Franco	45760639

# Contenido

Procedimientos Y Funciones .....	3
Capítulo I: Introducción.....	3
Capítulo II: Conceptos Fundamentales.....	3
Capítulo III: Procedimientos Almacenados .....	4
Capítulo IV: Inserción Masiva y Pruebas CRUD .....	5
Capítulo V: Funciones Almacenadas .....	7
Capítulo VI: Conclusiones.....	9
Optimización por Índices.....	9
Introducción .....	9
Tipos de índices.....	9
Otros tipos de índices;.....	10
Resultados de las pruebas.....	11
Evaluación de resultados.....	14
Conclusión .....	14
Triggers .....	14
Capítulo I: Introducción.....	14
Capítulo II: Tipos de Triggers .....	14
Capítulo III: Implementación y Pruebas .....	15
Capitulo IV: Evaluacion de Resultados.....	19
Capitulo V: Conclusiones .....	19
Bibliografía: .....	20

# Procedimientos Y Funciones

## Capítulo I: Introducción

En este capítulo se estudian dos elementos fundamentales para la lógica de negocio dentro de SQL Server: los **Procedimientos Almacenados** y las **Funciones Almacenadas**.

Ambos permiten centralizar operaciones, mejorar la seguridad, favorecer la reutilización de código y optimizar el rendimiento.

Este trabajo tiene como objetivo aplicar dichos elementos al proyecto **GestAcad**

## Capítulo II: Conceptos Fundamentales

### 2.1. Procedimiento Almacenado

Un **procedimiento almacenado** es un conjunto de instrucciones SQL precompiladas que se ejecutan en el servidor.

Pueden recibir parámetros, ejecutar operaciones complejas y retornar valores por parámetros de salida o mediante SELECT.

**Usos principales:** - Reglas de negocio - Validación de datos - Operaciones CRUD estandarizadas  
- Optimización de rendimiento

### 2.2. Función Almacenada

Una **función almacenada** retorna siempre un valor (escalar o tabla).

Se utiliza dentro de consultas SELECT o en expresiones.

**Usos principales:** - Cálculos derivados - Valores compuestos - Consultas que requieren lógica reutilizable

### 2.3. Diferencias Entre Procedimientos y Funciones

#### Retorno de valores:

Procedimientos: pueden devolver o no un valor.

Funciones: siempre deben retornar un valor.

#### Uso en consultas:

Procedimientos: no pueden utilizarse dentro de un SELECT.

Funciones: sí pueden ser llamadas en un SELECT, WHERE u otras expresiones.

#### Modificación de datos:

Procedimientos: pueden realizar operaciones INSERT, UPDATE y DELETE.

Funciones: no pueden modificar datos; solo realizar cálculos o consultas.

#### Complejidad y propósito:

Procedimientos: se usan para tareas completas y lógicas de negocio.

Funciones: se enfocan en cálculos o valores derivados.

## Capítulo III: Procedimientos Almacenados

Los **procedimientos almacenados** desarrollados para este trabajo son:

### 3.1. Insertar Alumnos

```
CREATE PROCEDURE spInsertar_Alumno
(
    @nombre VARCHAR(200),
    @apellido VARCHAR(200),
    @fecha_nacimiento DATE,
    @dni INT,
    @email VARCHAR(200),
    @pass VARCHAR(200)
)
AS
BEGIN
    IF EXISTS (SELECT 1 FROM dbo.alumnos WHERE dni = @dni)
        PRINT 'ya existe un alumno con ese dni';
    ELSE
        INSERT INTO dbo.alumnos (nombre, apellido, fecha_nacimiento, dni, email, pass)
        VALUES (@nombre, @apellido, @fecha_nacimiento, @dni, @email, @pass);
END;
GO
```

### 3.2. Eliminar Alumnos

```
CREATE PROCEDURE spEliminar_Alumno
(
    @dni INT
)
AS
BEGIN
    IF EXISTS (SELECT 1 FROM dbo.alumnos WHERE dni = @dni)
        DELETE FROM dbo.alumnos WHERE dni = @dni;
    ELSE
```

```

PRINT 'no existe alumno con ese dni';

END;

GO

3.3. Modificar Alumnos

CREATE PROCEDURE spModificar_Alumno
(
    @dni INT,
    @nombre VARCHAR(200) = NULL,
    @apellido VARCHAR(200) = NULL,
    @email VARCHAR(200) = NULL,
    @pass VARCHAR(200) = NULL
)
AS
BEGIN
    IF EXISTS (SELECT 1 FROM dbo.alumnos WHERE dni = @dni)
        BEGIN
            UPDATE dbo.alumnos
            SET nombre = COALESCE(@nombre, nombre),
                apellido = COALESCE(@apellido, apellido),
                email = COALESCE(@email, email),
                pass = COALESCE(@pass, pass)
            WHERE dni = @dni;
        END
    ELSE
        PRINT 'no existe alumno con ese dni';
END;
GO

```

## Capítulo IV: Inserción Masiva y Pruebas CRUD

### 4.1. Inserción de Datos Directa vs Procedimientos

Se cargaron **20.000 alumnos** de los cuales:

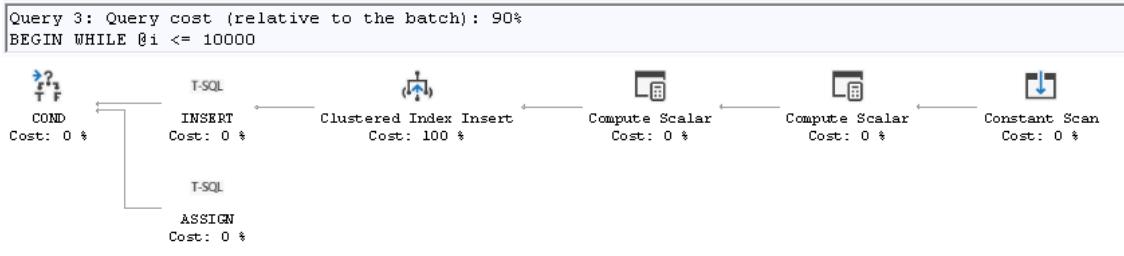
- 10.000 registros usando **inserción directa**

- 10.000 registros usando el procedimiento **spInsertar\_Alumno**

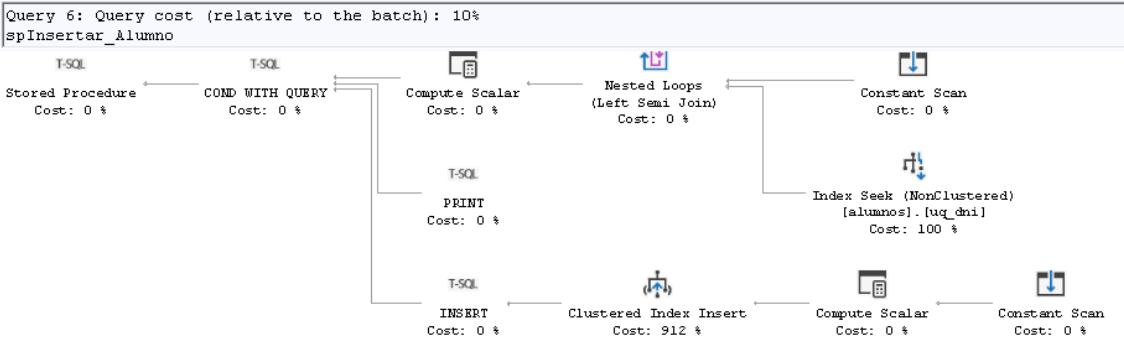
Con el objetivo de comparar la eficiencia de los diferentes métodos

Los resultados fueron:

### Inserción Directa



### Inserción Por Procedimiento



**Conclusión:** Al analizar los planes de ejecución, se observa que el bloque con inserciones directas tiene un costo relativo del 90%, mientras que la ejecución del procedimiento almacenado **spInsertar\_Alumno** tiene un costo del 10%. Esto sugiere que el procedimiento almacenado aprovecha mejor la optimización del motor SQL y reutiliza planes de ejecución, resultando más eficiente globalmente.

## 4.2. Pruebas de Eliminación y Modificación

Se realizaron pruebas con los procedimientos:

**spEliminar\_Alumno:** se utilizó para eliminar 5 alumnos

EXEC spEliminar\_Alumno 40000001

EXEC spEliminar\_Alumno 40000003

EXEC spEliminar\_Alumno 40000005

EXEC spEliminar\_Alumno 40000007

EXEC spEliminar\_Alumno 40000009

**resultado:** se borraron exitosamente los registros de los alumnos

**spModificar\_Alumno:** se utilizó para modificar distintos datos de 5 alumnos

EXEC spModificar\_Alumno 40000002, @nombre = 'Carlos'

```
EXEC spModificar_Alumno 40000004, @apellido = 'Rodriguez'  
EXEC spModificar_Alumno 40000006, @email = 'autos_845@hotmail.com'  
EXEC spModificar_Alumno 40000008, @pass = 'Contraseña0000008'  
EXEC spModificar_Alumno 40000010, @nombre = 'Juan', @apellido = 'Perez', @email =  
'juan_pe@hotmail.com', @pass = 'Contra10'
```

**resultado:** se modificaron exitosamente solo los datos modificados para los primeros cuatro casos y todos los campos modificables para el último caso

Por último se intento tanto eliminar como modificar los datos de un alumno que ya se había eliminado

```
EXEC spEliminar_Alumno 40000001
```

```
EXEC spModificar_Alumno 40000001
```

**resultado:** no hubo cambios en la tabla y se mostro en la pestaña de mensajes "no existe alumno con ese dni"

## Capítulo V: Funciones Almacenadas

Las **funciones almacenadas** desarrolladas para este trabajo son:

### 5.1. Nombre Completo del Alumno

```
CREATE FUNCTION fn_NombreYApellidoAlumno (@dni INT)
```

```
RETURNS VARCHAR(400)
```

```
AS
```

```
BEGIN
```

```
    DECLARE @nombreYApellido VARCHAR(400);
```

```
    SELECT @nombreYApellido = nombre + ' ' + apellido
```

```
    FROM alumnos
```

```
    WHERE dni = @dni;
```

```
    RETURN @nombreYApellido;
```

```
END;
```

```
GO
```

### 5.2. Estado del Alumno en la Carrera

```
CREATE FUNCTION fn_EstadoInscripcionCarrera (@dni INT, @id_carrera INT)
```

```
RETURNS VARCHAR(100)
```

```

AS
BEGIN

DECLARE @estado VARCHAR(100);
DECLARE @id_alumno INT;

SELECT @id_alumno = id_alumno FROM alumnos WHERE dni = @dni;

SELECT @estado = e.descripcion
FROM inscripcion_carrera ic
INNER JOIN estados e ON ic.id_estado = e.id_estado
WHERE ic.id_alumno = @id_alumno AND ic.id_carrera = @id_carrera;

RETURN @estado;
END;
GO

```

### **5.3. Promedio de Notas**

```

CREATE FUNCTION fn_PromedioNotas (@dni INT)
RETURNS FLOAT
AS
BEGIN

DECLARE @promedio FLOAT;
DECLARE @id_alumno INT;

SELECT @id_alumno = id_alumno FROM alumnos WHERE dni = @dni;

SELECT @promedio = AVG(calificacion)
FROM inscripcion_examen
WHERE id_alumno = @id_alumno AND calificacion IS NOT NULL;

RETURN @promedio;
END;

```

GO

## Capítulo VI: Conclusiones

Los procedimientos y las funciones almacenadas son herramientas clave para organizar y optimizar la lógica dentro de una base de datos. Permiten reutilizar código, mejorar el rendimiento y asegurar que las operaciones se ejecuten de manera consistente. Su uso contribuye a sistemas más claros, eficientes y fáciles de mantener.

# Optimización por Índices

## Introducción

En este trabajo se analiza el impacto que tienen los índices en el rendimiento de las consultas dentro de una base de datos. Para ello se realizaron distintas pruebas sobre una tabla que contiene un campo de tipo fecha, evaluando los tiempos de respuesta y los planes de ejecución antes y después de aplicar diferentes tipos de índices. El objetivo fue comprobar cómo el uso adecuado de estas estructuras permite optimizar la búsqueda de información, reduciendo el número de páginas leídas y mejorando la eficiencia del motor de base de datos.

## Tipos de índices

### Índice agrupado (Clustered)

Un índice agrupado define el orden físico en el que se almacenan las filas de una tabla según una o más columnas. Solo puede existir uno por tabla, ya que únicamente se puede ordenar físicamente de una manera. Al crear un índice agrupado sobre una columna, los datos se reordenan físicamente siguiendo los valores de dicha columna. Esto permite que, cuando se ejecuta una consulta por rangos (por ejemplo, entre fechas), el motor acceda directamente al punto inicial y lea secuencialmente las filas correspondientes, evitando un recorrido completo de la tabla.

Ventajas:

Mejora considerablemente el rendimiento de consultas que filtran por rangos.

Reduce la cantidad de lecturas necesarias (menos I/O).

Aprovecha el orden físico de la tabla.

Desventajas:

Solo puede haber uno por tabla.

Insertar o actualizar datos puede requerir reorganizar físicamente las páginas.

No es recomendable en columnas con valores repetidos o que cambian con frecuencia.

Código SQL para crear el índice

```
CREATE CLUSTERED INDEX idx_pruebas_fecha_agrupado ON Examen_Prueba(fecha);
```

Con este índice, la tabla Examen\_Prueba se organiza físicamente según la columna fecha, optimizando las búsquedas que filtran por períodos de tiempo.

### Índice no agrupado (Non-Clustered)

A diferencia del agrupado, el índice no agrupado no altera el orden físico de la tabla. Es una estructura independiente que contiene los valores de las columnas del índice junto con punteros a las filas reales de la tabla. Una tabla puede tener varios índices no agrupados, lo que permite optimizar consultas específicas que utilicen distintas columnas en sus condiciones.

Ventajas:

Permite crear varios índices por tabla.

Mejora el rendimiento de consultas que no pueden beneficiarse del índice agrupado.

Es posible usar la cláusula INCLUDE para agregar columnas adicionales y cubrir completamente la consulta, evitando acceder a la tabla base.

Desventajas:

Requiere espacio adicional.

Puede necesitar un paso adicional de búsqueda (lookup) para acceder a los datos reales.

Su mantenimiento implica más consumo de CPU e I/O al modificar la tabla.

Código SQL utilizado para crear el índice

```
CREATE NONCLUSTERED INDEX idx_pruebas_fecha_noagrupado ON Examen_Prueba(fecha)
INCLUDE (id_materia);
```

En este caso, el índice no agrupado utiliza fecha como clave principal e incluye id\_materia para cubrir las consultas que requieren ambas columnas, reduciendo el acceso directo a la tabla base.

Otros tipos de índices;

**Índices únicos:** garantizan que no existan valores duplicados en la columna o combinación de columnas especificadas. Son útiles cuando una columna debe tener valores distintos, como por ejemplo un número de documento o un correo electrónico.

**Índices compuestos:** combinan dos o más columnas como clave, mejorando el rendimiento en consultas que filtran o ordenan por más de un campo al mismo tiempo.

**Índices filtrados:** se crean sobre un subconjunto de filas que cumplen una condición específica. Son ideales cuando solo se necesita indexar una parte de la tabla (por ejemplo, registros activos o no nulos).

**Índices en columnas calculadas:** permiten indexar el resultado de una expresión o función, haciendo más rápidas las consultas que dependen de cálculos derivados.

**Índices columnstore:** almacenan los datos por columnas en lugar de filas, lo que los hace especialmente eficientes para consultas analíticas o de lectura intensiva en grandes volúmenes de datos (OLAP).

**Índices hash:** se utilizan en tablas optimizadas para memoria, permitiendo búsquedas extremadamente rápidas en valores exactos.

**Índices no agrupados optimizados para memoria:** también se aplican a tablas en memoria, pero almacenan referencias a las filas en estructuras tipo árbol, siendo útiles para búsquedas por rangos.

**Índices espaciales:** diseñados para trabajar con datos geográficos o espaciales, como coordenadas, polígonos o rutas.

**Índices XML:** permiten optimizar las consultas sobre columnas que almacenan datos en formato XML, mejorando la navegación dentro del contenido estructurado.

**Índices de texto completo:** se utilizan para realizar búsquedas eficientes dentro de textos largos, como descripciones o documentos, permitiendo localizar palabras o frases específicas dentro del contenido textual.

## Resultados de las pruebas

Para comprobar el efecto de los índices en el rendimiento, se realizaron cuatro escenarios: sin índice, con índice agrupado, con índice no agrupado y con índice no agrupado aplicado a ambas tablas. En cada caso se midieron los tiempos de respuesta y se analizaron los planes de ejecución del motor.

### Comparación sin índice agrupado

Figura 1. Comparación del rendimiento entre dos consultas sin índice agrupado. En esta prueba, el motor realizó un Table Scan, recorriendo la tabla completa para encontrar las filas que cumplen la condición. El tiempo de respuesta fue significativamente mayor y el número de lecturas lógicas elevado.

The screenshot shows two execution plans in SQL Server Management Studio. Both queries are SELECT \* FROM [Examen\_Pru] WHERE [fecha]>=@1 AND [fecha]<=@2.

**Consulta 1: Costo de la consulta (relativo al lote): 47 %**

```
SELECT * FROM [Examen_Pru] WHERE [fecha]>=@1 AND [fecha]<=@2
```

**Recorrido de tabla [Examen\_Pru]**  
Costo: 100 %  
0.063s  
24614 de  
24468 (100 %)

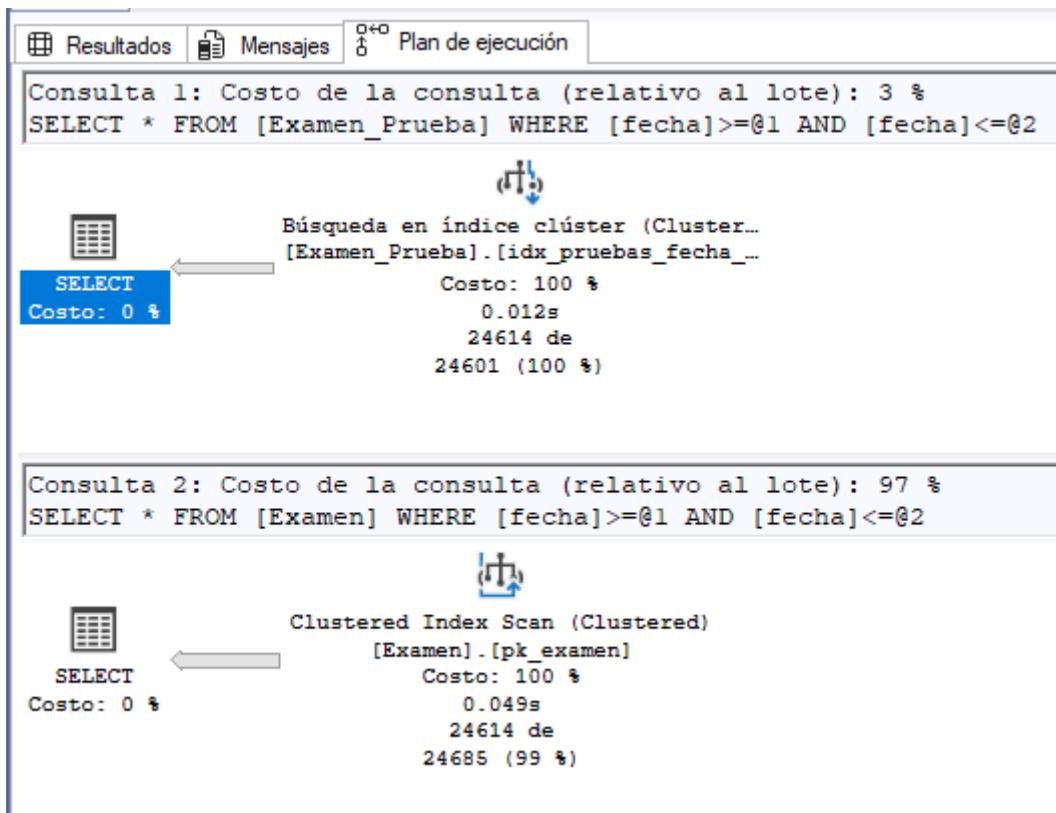
**Consulta 2: Costo de la consulta (relativo al lote): 53 %**

```
SELECT * FROM [Examen] WHERE [fecha]>=@1 AND [fecha]<=@2
```

**Clustered Index Scan (Clustered) [Examen].[pk\_examen]**  
Costo: 100 %  
0.049s  
24614 de  
24685 (99 %)

### Comparación con índice agrupado

Figura 2. Comparación del rendimiento tras aplicar un índice agrupado. La consulta se benefició del orden físico de la tabla, permitiendo un acceso secuencial eficiente mediante Index Seek. Se observó una notable disminución del tiempo de ejecución y del número de páginas leídas.



### Comparación con índice no agrupado

Figura 3. Comparación del rendimiento tras aplicar un índice no agrupado. El índice no agrupado mejoró la búsqueda sobre la columna fecha, reduciendo el tiempo respecto al heap inicial. Al incluir id\_materia, la consulta pudo resolverse directamente desde el índice, evitando la necesidad de lecturas adicionales en la tabla base.

100 % ✖ 2 ⚡ 0 ↑ ↓

Resultados Mensajes Plan de ejecución

Consulta 1: Costo de la consulta (relativo al lote): 97 %  
 SELECT [fecha], [id\_materia] FROM [Examen] WHERE [fecha]>=@1 AND [fecha]<=@2



```

  Clustered Index Scan (Clustered)
    [Examen].[pk_examen]
    Costo: 100 %
    0.057s
    24614 de
    24685 (99 %)
  
```

Consulta 2: Costo de la consulta (relativo al lote): 3 %  
 SELECT \* FROM [Examen\_Prueba] WHERE [fecha]>=@1 AND [fecha]<=@2



```

  Index Seek (NonClustered)
    [Examen_Prueba].[idx_pruebas_fecha_...]
    Costo: 100 %
    0.012s
    24614 de
    24601 (100 %)
  
```

#### Comparación con índice no agrupado en ambas tablas

Figura 4. Comparación del rendimiento con índices no agrupados aplicados a las dos tablas involucradas. Este escenario presentó el mejor resultado general, optimizando tanto la búsqueda principal como las uniones entre tablas. El plan de ejecución reflejó un acceso más eficiente y una reducción significativa en el costo total de la consulta.

100 % ✖ 2 ⚡ 0 ↑ ↓

Resultados Mensajes Plan de ejecución

Consulta 1: Costo de la consulta (relativo al lote): 47 %  
 SELECT [fecha], [id\_materia] FROM [Examen] WHERE [fecha]>=@1 AND [fecha]<=@2



```

  Index Seek (NonClustered)
    [Examen].[idx_examen_fecha_noagrupa...]
    Costo: 100 %
    0.011s
    24614 de
    24601 (100 %)
  
```

Consulta 2: Costo de la consulta (relativo al lote): 53 %  
 SELECT \* FROM [Examen\_Prueba] WHERE [fecha]>=@1 AND [fecha]<=@2



```

  Index Seek (NonClustered)
    [Examen_Prueba].[idx_pruebas_fecha_...]
    Costo: 100 %
    0.012s
    24614 de
    24601 (100 %)
  
```

## Evaluación de resultados

Las pruebas demostraron que la presencia de índices reduce drásticamente los tiempos de respuesta y la carga de lectura en disco. El índice agrupado resultó más beneficioso en consultas por rango de fechas, mientras que el no agrupado con columnas incluidas permitió optimizar consultas más específicas. En todos los casos, el cambio del plan de ejecución de Table Scan a Index Seek fue un indicador claro de la optimización lograda.

## Conclusión

El uso de índices en SQL resulta fundamental para mejorar el rendimiento de las consultas, especialmente en bases de datos con gran volumen de información. A través de las pruebas realizadas, se pudo comprobar cómo la creación de índices optimiza los tiempos de búsqueda y filtrado, mientras que su eliminación o mal uso puede generar un impacto negativo. Entender cuándo y dónde aplicar cada tipo de índice es esencial para mantener un equilibrio entre velocidad y eficiencia en las operaciones del sistema.

# Triggers

## Capítulo I: Introducción

Un Trigger (o disparador) es un objeto especial dentro de la base de datos, similar a un procedimiento almacenado, que se ejecuta de forma automática cuando ocurre un evento específico. Estos eventos son generalmente operaciones de manipulación de datos (DML) como INSERT, UPDATE o DELETE sobre una tabla específica.

El objetivo de este capítulo es aplicar esta tecnología en el proyecto GestAcad para cumplir dos objetivos críticos de negocio y auditoría:

1. **Auditoría (Trazabilidad):** Mantener un registro histórico de todos los cambios y eliminaciones que sufren los datos sensibles en la tabla alumnos.
2. **Integridad (Veto):** Prevenir activamente la eliminación de registros en tablas maestras, específicamente en la tabla Materia, para evitar la pérdida de integridad referencial histórica.

Este trabajo documentará los tipos de triggers disponibles en SQL Server, la implementación de los dos casos mencionados y la evaluación de su impacto en el sistema.

## Capítulo II: Tipos de Triggers

En SQL Server, los triggers se pueden clasificar según el evento que los dispara (DML, DDL) o el momento en que se ejecutan (AFTER, INSTEAD OF). Para este proyecto, nos centramos en los **Triggers DML**.

### 2.1. Triggers DML: AFTER (Después de)

Este es el tipo de trigger más común. Se ejecuta *después* de que la operación DML (INSERT, UPDATE o DELETE) se haya completado con éxito.

- **Caso de uso principal:** Auditoría y registro (logging).
- **Funcionamiento:** La acción (ej. el DELETE) ya ocurrió. El trigger se activa para registrar ese hecho en una tabla de bitácora. No puede cancelar la acción, solo reaccionar a ella.

- **Aplicación en GestAcad:** Se usará para implementar la **Tarea 1 (Auditoría)** en la tabla alumnos.

## 2.2. Triggers DML: INSTEAD OF (En lugar de)

Este trigger es más complejo y potente. Se ejecuta *en lugar de* la operación DML. Es decir, intercepta el comando (INSERT, UPDATE o DELETE) y **evita que se ejecute** en la tabla. En su lugar, ejecuta el código que se defina dentro del trigger.

- **Caso de uso principal:** Prevenir o vetar acciones, y permitir la actualización de vistas complejas (que no son actualizables por sí mismas).
- **Funcionamiento:** El DELETE original nunca se ejecuta. El trigger toma el control total.
- **Aplicación en GestAcad:** Se usará para implementar la **Tarea 2 (Veto)** en la tabla Materia.

## 2.3. Las Tablas Virtuales: inserted y deleted

Para que los triggers DML funcionen, SQL Server proporciona dos tablas virtuales (en memoria) que contienen las filas afectadas por la operación:

- **inserted:** Contiene las **filas nuevas**.
  - En un INSERT: Contiene los nuevos registros.
  - En un UPDATE: Contiene los datos *después* del cambio (valores nuevos).
- **deleted:** Contiene las **filas antiguas**.
  - En un DELETE: Contiene los registros que se acaban de borrar.
  - En un UPDATE: Contiene los datos *antes* del cambio (valores viejos).

Usaremos la tabla deleted en nuestras dos implementaciones.

## Capítulo III: Implementación y Pruebas

A continuación, se detalla la implementación y los resultados de las pruebas para cada trigger solicitado.

### 3.1. Tarea 1: Trigger de Auditoría en alumnos (AFTER)

**Objetivo:** Registrar en una tabla auditoria\_alumnos los valores *antiguos* de un alumno cuando este sea modificado (UPDATE) o eliminado (DELETE).

#### 3.1.1. Creación de la Tabla de Auditoría

Primero, se crea la tabla auxiliar:

```
CREATE TABLE auditoria_alumnos (
    id_auditoria INT IDENTITY(1,1) PRIMARY KEY,
    id_alumno INT NOT NULL,
    nombre_anterior VARCHAR(200),
```

```

apellido_anterior VARCHAR(200),
dni_anterior INT,
email_anterior VARCHAR(200),
operacion CHAR(1) NOT NULL CHECK (operacion IN ('U', 'D')),
usuario_bd VARCHAR(100) NOT NULL,
fecha_hora DATETIME NOT NULL
);

```

### **3.1.2. Creación del Trigger AFTER**

```

CREATE TRIGGER trg_audit_alumnos
ON alumnos -- Tabla vigilada
AFTER UPDATE, DELETE
AS
BEGIN
SET NOCOUNT ON;

DECLARE @usuario_bd VARCHAR(100) = SUSER_SNAME();
DECLARE @fecha_hora DATETIME = GETDATE();
DECLARE @tipo_operacion CHAR(1);

IF EXISTS (SELECT 1 FROM inserted)
    SET @tipo_operacion = 'U'; -- Fue un Update
ELSE
    SET @tipo_operacion = 'D'; -- Fue un Delete

-- Insertamos en la auditoría los datos VIEJOS desde la tabla 'deleted'
INSERT INTO auditoria_alumnos (
    id_alumno,
    nombre_anterior,
    apellido_anterior,
    dni_anterior,
    email_anterior,
    operacion,
    usuario_bd,
    fecha_hora
)
SELECT
    id,
    nombre,
    apellido,
    dni,
    email,
    @tipo_operacion,
    @usuario_bd,
    @fecha_hora
FROM deleted;

```

```

operacion,
usuario_bd,
fecha_hora
)
SELECT
d.id_alumno,
d.nombre,
d.apellido,
d.dni,
d.email,
@tipo_operacion,
@usuario_bd,
@fecha_hora
FROM
deleted d; -- Usamos la tabla mágica 'deleted'
END;

```

### **3.1.3. Pruebas y Resultados**

Se ejecutaron las siguientes operaciones:

-- 1. Insertamos un alumno de prueba

```

INSERT INTO alumnos (nombre, apellido, fecha_nacimiento, dni, email, pass)
VALUES ('Alumno', 'DePrueba', '2000-01-01', 99999999, 'prueba@email.com', 'pass123');

```

-- 2. Hacemos un UPDATE

```
UPDATE alumnos
```

```

SET email = 'prueba.modificado@email.com', nombre = 'AlumnoModificado'
WHERE dni = 99999999;
```

-- 3. Hacemos un DELETE

```
DELETE FROM alumnos
```

```
WHERE dni = 99999999;
```

Resultado en auditoria\_alumnos: Al consultar SELECT \* FROM auditoria\_alumnos;, la prueba es exitosa. Se observan dos registros:

	id_auditoria	id_alumno	nombre_anterior	apellido_anterior	email_anterior	dni_anterior	operacion	usuario_bd	fecha_hora
1	1	1	Alumno	DePrueba	prueba@email.com	99999999	U	HP15\Usuario	2025-11-03 20:57:04.617
2	2	1	AlumnoModificado	DePrueba	prueba.modificado@email.com	99999999	D	HP15\Usuario	2025-11-03 20:57:04.617

### 3.2. Tarea 2: Trigger de Veto en Materia (INSTEAD OF)

**Objetivo:** Prevenir la eliminación (DELETE) de registros en la tabla Materia y notificar al usuario.

#### 3.2.1 Creación del Trigger INSTEAD OF

```
CREATE TRIGGER trg_veto_delete_materia
```

```
ON Materia -- Tabla vigilada
```

```
INSTEAD OF DELETE
```

```
AS
```

```
BEGIN
```

```
    SET NOCOUNT ON;
```

```
-- 1. Emitir un mensaje de error claro al usuario.
```

```
    RAISERROR (
```

```
        'OPERACIÓN NO PERMITIDA: Las materias no pueden ser eliminadas (Trigger  
        trg_veto_delete_materia).',
```

```
        16, -- Nivel de severidad (error)
```

```
        1 -- Estado
```

```
    );
```

```
-- 2. Revertir la transacción.
```

```
    ROLLBACK TRANSACTION;
```

```
END;
```

#### 3.2.2 Pruebas y Resultados

Se ejecutaron las siguientes operaciones:

```
-- 1. Insertamos una materia de prueba
```

```
INSERT INTO Materia (nombre_materia) VALUES ('MateriaDePrueba');
```

```
-- (Asumamos que el ID generado es 4)
```

-- 2. Intentamos borrar la materia

```
DELETE FROM Materia WHERE id_materia = 4;
```

Resultado en la consola: La prueba es exitosa. La operación DELETE no se ejecuta y la consola de SQL Server devuelve el error que definimos:

```
Messages
--- Prueba 2: Trigger de Veto (trg_veto_delete_materia) ---
Insertando materia de prueba...

(1 row affected)
Intentando borrar la materia de prueba (ID: 4)...

(0 rows affected)
PRUEBA EXITOSA: El trigger de veto funcionó.
Mensaje de error capturado: OPERACIÓN NO PERMITIDA: Las materias no pueden ser eliminadas (Trigger trg_veto_delete_materia).
    Consideré marcarlas como "inactivas" en lugar de borrarlas.
VERIFICACIÓN: La materia (ID: 4) sigue existiendo en la BD.
Limpiando materia de prueba (deshabilitando trigger temporalmente)...

(1 row affected)
Limpieza completada y trigger rehabilitado.

Completion time: 2025-11-10T15:02:51.5484115-03:00
```

## Capítulo IV: Evaluacion de Resultados

La implementación de los triggers cumple con los criterios de evaluación establecidos:

**Impacto en la Integridad:** El trigger `trg_veto_delete_materia` (INSTEAD OF) garantiza la integridad de los datos a un nivel que una restricción de FOREIGN KEY por sí sola no puede. Previene activamente una acción de negocio (borrar materias) que se considera peligrosa, evitando así borrados en cascada accidentales y la pérdida de datos históricos.

**Impacto en la Seguridad:** El trigger `trg_audit_alumnos` (AFTER) incrementa exponencialmente la seguridad del sistema al implementar la trazabilidad. Ahora es posible responder a las preguntas "quién, qué y cuándo" se modificó o borró un dato crítico de un alumno, lo cual es fundamental para cualquier sistema académico.

**Funcionalidad:** Como se demostró en el Capítulo III, las pruebas confirman que ambos triggers son 100% funcionales y responden correctamente a los eventos UPDATE, DELETE.

## Capítulo V: Conclusiones

Los triggers demostraron ser una herramienta extremadamente poderosa para centralizar la lógica de negocio y las reglas de auditoría directamente en el motor de la base de datos.

### 5.1 FUNCIONALIDAD:

Se comprobó que los triggers AFTER son una solución robusta y confiable para implementar auditorías a nivel de base de datos. Se demostró que los triggers INSTEAD OF son una herramienta efectiva para implementar reglas de negocio restrictivas (veto), impidiendo operaciones no deseadas que las restricciones (constraints) simples no podrían manejar.

### 5.2 IMPACTO EN LA INTEGRIDAD Y SEGURIDAD:

La auditoría (Tarea 1) incrementa masivamente la SEGURIDAD y la TRAZABILIDAD. Ahora es posible responder "quién, qué y cuándo" se modificó un dato crítico. El veto (Tarea 2) garantiza la INTEGRIDAD de los datos al prevenir el borrado de entidades maestras (como 'Materia'), evitando así borrados en cascada accidentales o la pérdida de datos históricos.

### **5.3 CONSIDERACIONES (DIFICULTADES/VENTAJAS):**

-VENTAJA (Centralización y Robustez): Las reglas de negocio (como "no borrar materias") están en la BD, no en la aplicación. Esto asegura que la regla se cumpla sin importar quién se conecte (un desarrollador, un administrador, una app web, etc.). La auditoría AFTER es automática y no puede ser "olvidada" por un programador de aplicaciones.

DESVENTAJA (Riesgo): Los triggers pueden ser "invisibles" para los desarrolladores. Un programador podría no entender por qué su DELETE falla o por qué la base de datos se vuelve lenta, ya que la lógica está "oculta" en el trigger. Una documentación clara es esencial.

-DESVENTAJA (Rendimiento): Un trigger se ejecuta *por cada* operación. Si el trigger es complejo (ej. consulta muchas tablas), puede impactar severamente el rendimiento (performance) de las operaciones DML en tablas muy concurridas. Deben ser lo más eficientes posible.

## **Bibliografía:**

Triggers DML (AFTER / INSTEAD OF) (SQL Server). <https://learn.microsoft.com/es-es/sql/relational-databases/triggers/dml-triggers>

CREATE TRIGGER (Transact-SQL). <https://learn.microsoft.com/es-es/sql/t-sql/statements/create-trigger-transact-sql>

Usar las tablas inserted y deleted. <https://learn.microsoft.com/es-es/sql/relational-databases/triggers/use-the-inserted-and-deleted-tables>

Páginas visitadas el 03/11/25

<https://learn.microsoft.com/es-es/sql/relational-databases/sql-server-index-design-guide> <https://learn.microsoft.com/es-es/sql/relational-databases/indexes/indexes?view=sql-server-ver16> Páginas visitada el 9/11/2025.