



# Bases de Datos I

## Proyecto de Estudio

"Sistema de Gestión para Tiendas de Ropas"

## Autores

- Gariglio, Nestor David
- Gonzalez Billordo, Abel Benjamin
- Silva Zeniquel, Pablo
- Vera, Pablo Gabriel

## Profesores a Cargo

- Lic. Villegas, Dario O.
- Exp. Cuzziol, Juan J.
- Lic. Vallujos, Walter O.
- Lic. Badaracco Numa

## Universidad

"Facultad de Ciencias Exactas y Naturales y Agrimensura"

## Carrera

"Licenciatura en Sistemas de la Información"

# **Índice**

## **Capítulo I – Introducción**

- 1.1 Tema: Sistema de Gestión para una tienda de ropa
- 1.2 Definición del Problema
- 1.3 Objetivos
  - 1.3.1 Objetivos General
  - 1.3.2 Objetivos Específicos

## **Capítulo II – Marco Conceptual**

- 2.1 Modelo Relacional y la Plataforma de Implementación
- 2.2 Integridad y Consistencia de los Datos
  - 2.2.1 Integridad Referencial
  - 2.2.2 Propiedades ACID y Transacciones
- 2.3 Componentes Programables del Servidor (Procedimientos y Funciones)
- 2.4 Optimización de Consultas e Indexación

## **Capítulo III – Metodología Seguidas**

- 3.1 Descripción de cómo se realizó el Trabajo Práctico
- 3.2 Herramientas (Instrumentos y procedimientos)

## **Capítulo IV – Desarrollo del tema**

- 4.1 Modelo de Datos Relacionales
- 4.2 Diccionario de Datos
  - 4.2.1 Tabla 1: proveedor
  - 4.2.2 Tabla 2: categoría
  - 4.2.3 Tabla 3: producto
  - 4.2.4 Tabla 4: ticket
  - 4.2.5 Tabla 5: detalle\_ticket
  - 4.2.6 Tabla 6: pago
  - 4.2.7 Tabla 7: usuario
  - 4.2.8 Tabla 8: tipo\_pago
  - 4.2.9 Tabla 9: rol
- 4.3 Script Base de datos y Lote de datos de "NP"
- 4.4 Procedimientos-Funciones Almacenadas

- 4.4.1 Procedimientos Almacenados
  - 4.4.2 Utilización
  - 4.4.3 Funciones Almacenadas
  - 4.4.4 Utilización
  - 4.4.5 Diferencia entre procedimiento y función
  - 4.4.6 Aplicación en operaciones CRUD
  - 4.4.7 Ingreso de lote de dato (Directa - Procedimental)
  - 4.4.8 Aplicación de update y Delete
  - 4.4.9 Aplicación en registro funciones almacenada
  - 4.4.10 Comparación de eficiencia
  - 4.4.11 Análisis de rendimiento - eficiencia.
  - 4.4.12 Herramientas de Medición
  - 4.4.13 Caso de Prueba 1 : Inserción de Lote de Datos
  - 4.4.14 Caso de Prueba 2 : Calculo de monto
  - 4.4.15 Criterios de Comparación
- 4.5 Introducción y Fundamentación Teórica
    - 4.5.1 La Problemática del Rendimiento
    - 4.5.2 Importancia de los Índices
    - 4.5.3 Tipos de Índices Utilizados y Conceptos Clave
    - 4.5.4 Desarrollo Práctico: Escenario de Prueba
  - 4.6 Manejo de Transacciones
    - 4.6.1 Definición: ¿Qué es una Transacción?
    - 4.6.2 El Propósito: Las Propiedades ACID
    - 4.6.3 Aplicación en la Base de Datos "tienda\_ropa"
    - 4.6.4 Script de Transacción Exitosa (COMMIT)
    - 4.6.5 Script de Transacción Fallida (ROLLBACK)
    - 4.6.6 Script (Script-Venta-Fallida.sql)
  - 4.7 Índices Columnares en SQL Server
    - 4.7.1 ¿Qué son los índices columnares?
    - 4.7.2 Ventajas y su fundamento técnico
      - Rendimiento superior en consultas analíticas
      - Reducción del tamaño de almacenamiento
      - Lectura mucho más eficiente de columnas específicas
      - Segment elimination

- Procesamiento vectorizado (Batch Mode)
- 4.7.3 Desventajas y explicación técnica
  - Peor rendimiento en cargas pequeñas y operaciones OLTP
  - Requiere más memoria para consultas complejas
  - No es adecuado para tablas altamente transaccionales
- 4.7.4 ¿Cómo utilizar los índices columnares?
  - Tipo 1
  - Tipo 2
  - Aplicación práctica en el proyecto
  - Creación del índice columnar usado en pruebas
  - Análisis de rendimiento – Comparación real obtenida
  - Consultas ejecutadas
  - Resultados obtenidos

## **Capítulo V – Conclusiones**

- 5.1 Conclusiones Procedimientos y funciones almacenadas
  - 5.1.1 Conclusión General de Eficiencia
- 5.2 Conclusiones Optimización de Consultas (Índices)
  - 5.2.1 Análisis de Resultados (Lectura)
  - 5.2.3 Interpretación:
  - 5.2.4 Análisis de Impacto en Escritura (INSERT)
  - 5.2.5 Conclusión
- 5.3 Conclusiones Manejo de Transacciones
  - 5.3.1 Manejo de Transacciones
  - 5.3.2 Las Transacciones como Requisito de Negocio
  - 5.3.3 Garantía de Atomicidad (ACID)
  - 5.3.4 Conclusión Final
- 5.4 Conclusiones Índices Columnares
  - 5.4.1 Análisis de Resultados (Lectura)
- Bibliográfica

# Bases de Datos I

## Capítulo I - Introducción

### 1.1 Tema: Sistema de Gestión para una tienda de ropa

En la actualidad, las pequeñas y medianas tiendas de indumentaria suelen enfrentar dificultades en la organización y control de sus procesos internos. La gestión manual de tareas como el control del inventario, el registro de ventas o la solicitud a proveedores genera retrasos, errores humanos y falta de información confiable para la toma de decisiones.

Frente a esta situación, se propone como grupo el desarrollo de un **Sistema de Gestión para una Tienda de Ropa**, cuyo propósito es optimizar de forma integral las operaciones de la tienda mediante la automatización de actividades administrativas y comerciales. Con ello, se busca reemplazar métodos manuales ineficientes por una herramienta sistematizada que brinde orden, agilidad y confiabilidad en el manejo de datos.

Este proyecto se enmarca dentro de la formación académica en bases de datos, aportando una solución práctica que atiende necesidades reales de organización en el ámbito comercial.

### 1.2 Definición del Problema

La gestión tradicional en tiendas de ropa presenta los siguientes inconvenientes:

- Errores frecuentes en el control del inventario y en el registro de ventas.
- Pérdida de tiempo en tareas administrativas repetitivas.
- Carencia de información precisa y actualizada que permita tomar decisiones oportunas.
- Falta de mecanismos que faciliten la reposición de productos y la relación directa con proveedores.

Ante estas limitaciones surge el siguiente interrogante:

**¿De qué manera un sistema de gestión puede automatizar y optimizar las tareas administrativas y operativas de una tienda de ropa, reduciendo errores y mejorando la eficiencia en la administración del negocio?**

### 1.3 Objetivos

#### 1.3.1 Objetivo General

Desarrollar un **Sistema de Gestión para una Tienda de Ropa** que permita optimizar las tareas administrativas y comerciales mediante la automatización de procesos, brindando mayor eficiencia en la organización y facilitando la toma de decisiones a partir de información confiable y accesible.

#### 1.3.2 Objetivo Específicos

- Diseñar un módulo de gestión de prendas que permita organizar los artículos de la tienda de forma clara y ordenada.

- Implementar un sistema de ventas que registre operaciones de manera automática y facilite la emisión de comprobantes.
  - Incorporar un módulo de reportes y estadísticas que proporcione indicadores periódicos para el análisis del negocio.
  - Establecer alertas de stock que favorezcan la reposición oportuna de productos.
  - Desarrollar la gestión de proveedores con vinculación directa a los productos, posibilitando pedidos automáticos.
  - Brindar una interfaz sencilla e intuitiva que facilite su uso tanto a vendedores como a administradores
- 

## Capítulo II - Marco Conceptual

### 2.1 El Modelo Relacional y la Plataforma de Implementación

Este proyecto se fundamenta en el **Modelo Relacional** para la representación y gestión de los datos del sistema de control de stock. La elección de este modelo se basa en su probada solidez, flexibilidad y en la garantía de **Integridad de Datos** a través de la formalización de las relaciones entre entidades.

Como **Sistema Gestor de Bases de Datos (SGBD)**, se ha seleccionado **Microsoft SQL Server**. Esta elección está justificada por su robustez, su escalabilidad para manejar grandes volúmenes de datos y por ser la plataforma que soporta la implementación de **Índices Columnares**, uno de los temas técnicos asignados a la investigación.

### 2.2 Integridad y Consistencia de los Datos

La calidad de un sistema transaccional como el propuesto (Sistema de Gestión de Stock) depende directamente de la capacidad del SGBD para mantener la integridad y la consistencia de los datos.

#### 2.2.1 Integridad Referencial

La integridad referencial es garantizada por las **Claves Primarias (PK)** y **Claves Foráneas (FK)** definidas en el Modelo Relacional.

- **Claves Primarias:** Aseguran que cada fila en una tabla sea única.
- **Claves Foráneas:** Mantienen la coherencia en las relaciones entre tablas (asegura que cada registro de la tabla `Detalles_Ticket` apunte a un `Ticket` de venta existente y a un `Producto` válido, evitando datos huérfanos).

#### 2.2.2 Propiedades ACID y Transacciones

Las operaciones críticas del sistema, como el registro de una venta o la recepción de un proveedor, deben ser gestionadas como **Transacciones** para cumplir con el estándar **ACID**:

- **Atomicidad:** Una transacción se ejecuta completamente o no se ejecuta en absoluto. En una venta, si la inserción del `Ticket` falla, la actualización del `Stock` debe revertirse (rollback).
- **Consistencia:** La transacción lleva la base de datos de un estado válido a otro, manteniendo la integridad referencial y las reglas de negocio.

- **Aislamiento:** Múltiples transacciones concurrentes no deben interferir entre sí.
- **Durabilidad:** Una vez que la transacción es confirmada (`COMMIT`), los cambios son permanentes en el almacenamiento persistente.

El **Manejo de Transacciones** (Tema 3) y las **Transacciones Anidadas** se implementarán para asegurar la **Atomicidad** y la **Consistencia** en procesos multifase.

## 2.3 Componentes Programables del Servidor (Procedimientos y Funciones)

La lógica de negocio compleja y las operaciones de manipulación de datos son encapsuladas y centralizadas en el servidor de la base de datos mediante componentes programables.

- **Procedimientos Almacenados (PAs):** Son conjuntos de sentencias SQL precompiladas que residen en el servidor. Se utilizarán para las operaciones **CRUD** (`Insertar`, `Modificar`, `Borrar` registros), ya que mejoran la seguridad y reducen el tráfico de red, pues solo se envía la llamada al PA y no las sentencias SQL completas.
- **Funciones Almacenadas:** Son bloques de código que devuelven un valor. Se emplearán para realizar cálculos o retornar información específica (e.g., el cálculo de un precio final o la edad de un usuario a partir de su fecha de nacimiento), pudiendo ser llamadas dentro de sentencias SQL.

El **Tema 1** se enfocará en demostrar cómo estos componentes programables aumentan la eficiencia y el control sobre la manipulación de datos en el sistema de stock.

## 2.4 Optimización de Consultas e Indexación

La optimización de consultas es crítica para la performance de cualquier sistema de información. La base de esta optimización es la **Indexación**.

- **Índices Tradicionales (Orientados a Fila):** Los índices son estructuras de datos que aceleran la búsqueda de registros en tablas voluminosas. Se implementarán **Índices Agrupados (Clustered) y No Agrupados (Non-Clustered)** (Tema 2) en tablas con alta frecuencia de consulta (como `Productos` o `Tickets`) para reducir drásticamente los tiempos de respuesta. Las pruebas compararán el **Plan de Ejecución** de las consultas antes y después de aplicar los índices para cuantificar la mejora de rendimiento.
- **Índices Columnares (SQL Server):** A diferencia de los índices tradicionales que almacenan datos fila por fila (ideales para transacciones), los índices columnares almacenan los datos columna por columna. Esta arquitectura está optimizada para tareas de **Data Warehousing** y consultas analíticas que procesan grandes agregaciones de datos. El **Tema 4** investigará la mejora de rendimiento que ofrecen estos índices en la tabla con **un millón de registros** para tareas de reportes y análisis, en contraposición con los índices orientados a fila.

## Capítulo III - Metodología Seguidas

En el presente capítulo se detalla la metodología de trabajo adoptada por el equipo para el desarrollo del proyecto. Se describen las herramientas tecnológicas seleccionadas, la organización de las tareas, el flujo de trabajo seguido y las distintas fases que compusieron la ejecución del plan

de trabajo, desde la implementación de la base de datos hasta la elaboración de la documentación final.

### **3.1 Descripción de cómo se realizó el Trabajo Práctico**

El proyecto se abordó siguiendo un esquema de trabajo secuencial y colaborativo, basado en la planificación inicial.

La división de tareas fue asignada a miembros específicos del equipo desde el comienzo. El flujo de trabajo se estructuró de manera que existían dependencias entre las tareas: la finalización de la implementación del DDL por parte del alumno era un requisito indispensable para que el alumno siguiente pudiera comenzar con la carga de datos (DML).

Una vez que el alumno finalizó el script creacion\_tablas.sql, lo subió al repositorio de GitHub. A partir de ese momento, el alumno siguiente pudo acceder a la estructura final de la base de datos para desarrollar el script carga\_datos\_base.sql. La plataforma GitHub fue fundamental para asegurar que todos los miembros tuvieran acceso a la última versión de los archivos.

### **3.2 Herramientas (Instrumentos y procedimientos)**

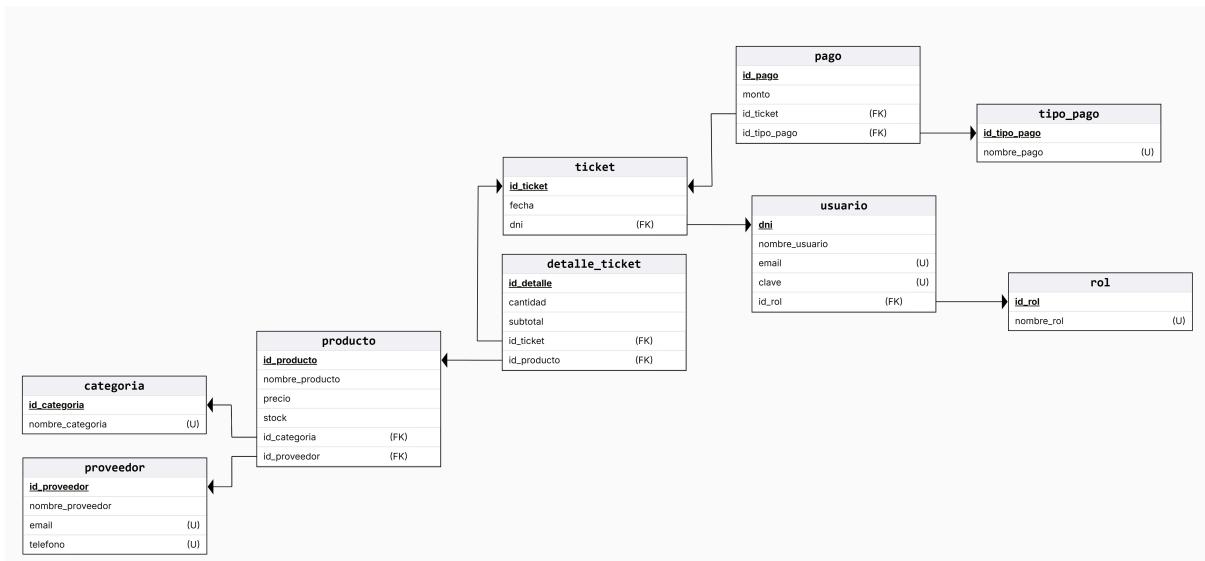
A continuación, se describen los instrumentos y procedimientos utilizados para el diseño, implementación, tratamiento de la información y gestión del proyecto.

- ERDPlus: Herramienta de modelado en línea utilizada en la fase de diseño para la creación del Modelo Entidad-Relación y la generación del diccionario de datos.
- SQL Server Management Studio (SSMS): Herramienta principal utilizada para la implementación, gestión y prueba de los comandos SQL, incluyendo la creación de usuarios, configuración de permisos y la ejecución de procedimientos.
- Git y GitHub: Se utilizó Git como sistema de control de versiones para gestionar los scripts SQL y la documentación del proyecto. La plataforma GitHub se usó como repositorio centralizado para el trabajo en equipo.
- Consultas SQL y Scripts: Se desarrollaron scripts para automatizar la creación de la estructura de la base de datos (DDL) y la carga de datos inicial (DML). Además, se escribieron los procedimientos almacenados y funciones para facilitar la gestión de los datos de la tienda de ropa.
- Método de Revisión Bibliográfica: La consulta de documentación oficial de Microsoft SQL Server y guías en Internet fue fundamental para comprender las mejores prácticas y aplicarlas correctamente en cada etapa del proyecto.
- Herramientas de Comunicación: Se utilizó WhatsApp y Notion para la coordinación diaria y Google Meet y Discord para las reuniones de planificación y seguimiento del equipo.

---

## **Capítulo IV - Desarrollo del tema**

### **4.1 Modelo de Datos Relacionales**



## 4.2 Diccionario de Datos

### 4.2.1 Tabla 1: proveedor

Características de la tabla			
Nombre	proveedor		
modulo	ventas		
descripción	esta tabla se diseña para almacenar los datos del proveedor de productos		
Características de los datos			
campo	tipo	long	significado
<code>id_proveedor</code>	int	11	identificación única para proveedor
<code>nombre_proveedor</code>	varchar	150	nombre del proveedor
<code>email</code>	varchar	150	email del proveedor
<code>teléfono</code>	int	11	teléfono del proveedor
Restricciones			
campo	tipo restricción		
<code>id_proveedor</code>	PRIMARY KEY		
<code>email</code>	UNIQUE		
<code>teléfono</code>	UNIQUE		
Claves Foráneas			
campo	entidad asociada		
-	-		

### 4.2.2 Tabla 2: categoría

Características de la tabla			
nombre	categoría		
modulo	ventas		
descripción	esta tabla se diseñó para almacenar los datos de categoría de productos		
Características de los datos			
campo	tipo	long	Significado
id_categoria	int	11	identificación única para categoría
nombre_categoria	varchar	150	nombre de categoría de producto
Restricciones			
Campo	tipo restricción		
id_categoria	PRIMARY KEY		
Claves Foráneas			
campo	entidad asociada		
-	-		

#### 4.2.3 Tabla 3: producto

Características de la tabla			
nombre	producto		
modulo	Ventas		
descripción	esta tabla se diseñó para almacenar los datos de productos		
Características de los datos			
campo	tipo	long	significado
id_producto	int	11	identificación única para producto
nombre_producto	varchar	150	nombre del producto
precio	float	-	precio del producto
stock	int	11	stock del producto
id_categoria	int	11	identificación única para categoría
id_proveedor	int	11	identificación única para proveedor
Restricciones			
Campo	tipo restricción		
id_proveedor	PRIMARY KEY		
Claves Foráneas			
campo	entidad asociada		
id_categoria	categoria		
id_proveedor	proveedor		

#### 4.2.4 Tabla 4: ticket

Características de la tabla			
nombre	ticket		
modulo	ventas		
descripción	esta tabla se diseñó para almacenar los datos de cada ticket		
Características de los datos			
campo	tipo	long	significado
id_ticket	int	11	identificación única para ticket
fecha	date	-	fecha de emisión del ticket
dni	int	11	dni del usuario
Restricciones			
Campo	tipo restricción		
id_ticket	PRIMARY KEY		
Claves Foráneas			
campo	entidad asociada		
dni	usuario		

#### 4.2.5 Tabla 5: detalle\_ticket

Características de la tabla			
nombre	detalle_ticket		
modulo	ventas		
descripción	esta tabla se diseñó para registrar los detalles que corresponden al ticket		
Características de los datos			
campo	tipo	long	significado
id_detalle	int	11	identificación única para detalle_ticket
cantidad	int	11	refleja la cantidad de un producto
subtotal	float	-	refleja el monto de una línea del ticket
id_ticket	int	11	identificación única para ticket
id_producto	int	11	identificación única para producto
Restricciones			
campo	tipo restricción		
id_detalle	PRIMARY KEY		
Claves Foráneas			
Campo	entidad asociada		
id_ticket	ticket		
id_producto	producto		

#### 4.2.6 Tabla 6: pago

Características de la tabla			
nombre	pago		
modulo	ventas		
descripción	esta tabla se diseñó para almacenar los datos que corresponden al pago		
Características de los datos			
campo	tipo	long	significado
id_pago	int	11	identificación única para pago
monto	float	-	refleja el monto a pagar
id_ticket	int	11	identificación única para ticket
id_tipo_pago	int	11	identificación única para tipo_pago
Restricciones			
campo	tipo restricción		
id_pago	PRIMARY KEY		
Claves Foráneas			
Campo	entidad asociada		
id_ticket	ticket		
id_tipo_pago	tipo_pago		

#### 4.2.7 Tabla 7: usuario

Características de la tabla			
nombre	usuario		
modulo	ventas		
descripción	esta tabla se diseñó para almacenar los datos que corresponden al usuario		
Características de los datos			
campo	tipo	long	significado
dni	int	11	identificación única para usuario
nombre_usuario	varchar	250	nombre del usuario
email	varchar	150	email del usuario
clave	varchar	100	clave del usuario
id_rol	int	11	identificación única para rol
Restricciones			
campo			tipo restricción
dni			PRIMARY KEY
email			UNIQUE
clave			UNIQUE
Claves Foráneas			
campo			entidad asociada
id_rol			rol

#### 4.2.8 Tabla 8: tipo\_pago

Características de la tabla			
nombre	tipo_pago		
modulo	ventas		
descripción	esta tabla se diseñó para almacenar los datos que corresponden al tipo de pago		
Características de los datos			
campo	tipo	long	significado
id_tipo_pago	int	11	identificación única para tipo de pago
nombre_pago	varchar	50	nombre del tipo de pago
Restricciones			
campo			tipo restricción
id_tipo_pago			PRIMARY KEY
Claves Foráneas			
campo			entidad asociada
-			-

#### 4.2.9 Tabla 9: rol

Características de la tabla			
nombre	rol		
modulo	ventas		
descripción	esta tabla se diseñó para almacenar los datos que corresponden al rol del usuario		
Características de los datos			
campo	tipo	long	significado
id_rol	int	11	identificación única para rol
nombre_rol	varchar	50	nombre del rol
Restricciones			
campo			tipo restricción
dni			PRIMARY KEY
nombre_rol			UNIQUE
Claves Foráneas			
campo			entidad asociada
-			-

---

## 4.3 Script Base de datos “NP”

[https://github.com/Vera-Pablo/Proyecto\\_Bases\\_de\\_Datos/blob/main/second\\_part/Etapa%201/Script-NP.sql](https://github.com/Vera-Pablo/Proyecto_Bases_de_Datos/blob/main/second_part/Etapa%201/Script-NP.sql)

### Script Lote de datos “NP”

[https://github.com/Vera-Pablo/Proyecto\\_Bases\\_de\\_Datos/blob/main/second\\_part/Etapa%201/LoteDatos-NP.sql](https://github.com/Vera-Pablo/Proyecto_Bases_de_Datos/blob/main/second_part/Etapa%201/LoteDatos-NP.sql)

---

## 4.4 Procedimientos-Funciones Almacenadas

### ¿Que son los procedimientos y funciones almacenadas?

Son bloques de códigos escritos una sola vez y guardados en la base de datos, permitiendo reutilizarlos cuando se requiera realizar una tarea específica donde el bloque de código puede devolver un valor o no.

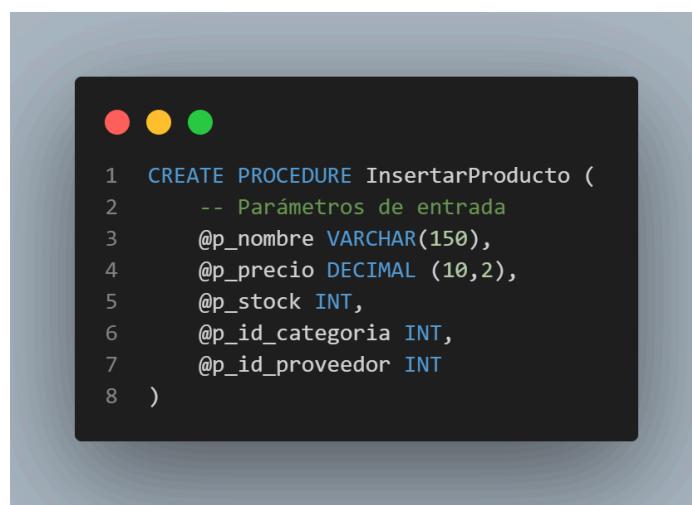
Facilitando la reutilización, mantenibilidad, legibilidad y rendimiento en las consultas de la base de dato en la que se trabaja.

---

### 4.4.1 Procedimientos Almacenados

Un procedimiento almacenado de SQL Server es un grupo de una o varias instrucciones Transact-SQL (funciones que permiten manipular datos). Los procedimientos se asemejan a las construcciones de otros lenguajes de programación, porque pueden:

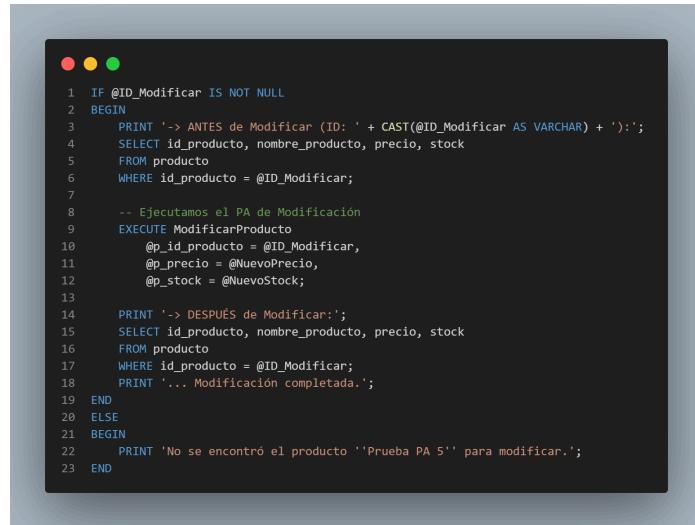
- Aceptar parámetros de entrada y devolver varios valores en forma de parámetros de salida al programa que realiza la llamada.



```
1 CREATE PROCEDURE InsertarProducto (
2     -- Parámetros de entrada
3     @p_nombre VARCHAR(150),
4     @p_precio DECIMAL (10,2),
5     @p_stock INT,
6     @p_id_categoria INT,
7     @p_id_proveedor INT
8 )
```

- 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.

Devuelve conjuntos de resultados, parámetros de salida o ambos, pero como tal un valor como una función.



```
1 IF @ID_Modificar IS NOT NULL
2 BEGIN
3     PRINT '-- ANTES de Modificar (ID: ' + CAST(@ID_Modificar AS VARCHAR) + ')';
4     SELECT id_producto, nombre_producto, precio, stock
5     FROM producto
6     WHERE id_producto = @ID_Modificar;
7
8     -- Ejecutamos el PA de Modificación
9     EXECUTE ModificarProducto
10    @p_id_producto = @ID_Modificar,
11    @p_precio = @NuevoPrecio,
12    @p_stock = @NuevoStock;
13
14    PRINT '-- DESPUÉS de Modificar:';
15    SELECT id_producto, nombre_producto, precio, stock
16    FROM producto
17    WHERE id_producto = @ID_Modificar;
18    PRINT '... Modificación completada.';
19 END
20 ELSE
21 BEGIN
22     PRINT 'No se encontró el producto ''Prueba PA 5'' para modificar.';
23 END
```

#### 4.4.2 ¿Como se utiliza?

Cuando queremos mejorar el rendimiento al reducir el tráfico de red (reducir cantidad de datos que se envían), aumentar la seguridad al controlar el acceso a los datos y facilitar el mantenimiento al permitir reutilizar código o modularizar la lógica, entra en acción ese mecanismo.

#### Creación de Procedimientos Almacenados

Aclaración: En este artículo se describe cómo crear un procedimiento almacenado de SQL Server mediante SQL Server Management

1. En SSMS, conéctese a una instancia de SQL Server.
2. Seleccione **Nueva consulta** en la barra de herramientas.
3. Escriba el código siguiente en la ventana Consulta, reemplazando `<ProcedureName>`, los nombres y los tipos de datos de cualquier parámetro y la instrucción SELECT por sus propios valores.

```
CREATE PROCEDURE <ProcedureName>
    @<ParameterName1> <data type>,
    @<ParameterName2> <data type>
AS

    SET NOCOUNT ON;
    SELECT <your SELECT statement>;
    GO
```

4. Seleccione **Ejecutar** en la barra de herramientas para ejecutar la consulta. Se creará el procedimiento almacenado.

#### Ejecución del Procedimiento Almacenado

Aclaración: En este artículo se describe cómo crear un procedimiento almacenado de SQL Server mediante SQL Server Management

1. En SSMS, conéctese a una instancia de SQL Server o Azure SQL Database.
2. En la barra de herramientas, seleccione **Nueva consulta**.
3. Escriba una instrucción EXECUTE con la siguiente sintaxis en la ventana Consulta y proporcione valores para todos los parámetros esperados:

```
EXECUTE <ProcedureName> N'<Parameter 1 value>, N'<Parameter x value>;  
GO
```

Por ejemplo, la siguiente instrucción de Transact-SQL ejecuta el procedimiento almacenado `uspGetCustomerCompany` y con `Cannon` como valor del parámetro `@LastName` y `Chris` como valor del parámetro `@FirstName`:

```
EXEC SalesLT.uspGetCustomerCompany N'Cannon', N'Chris';  
GO
```

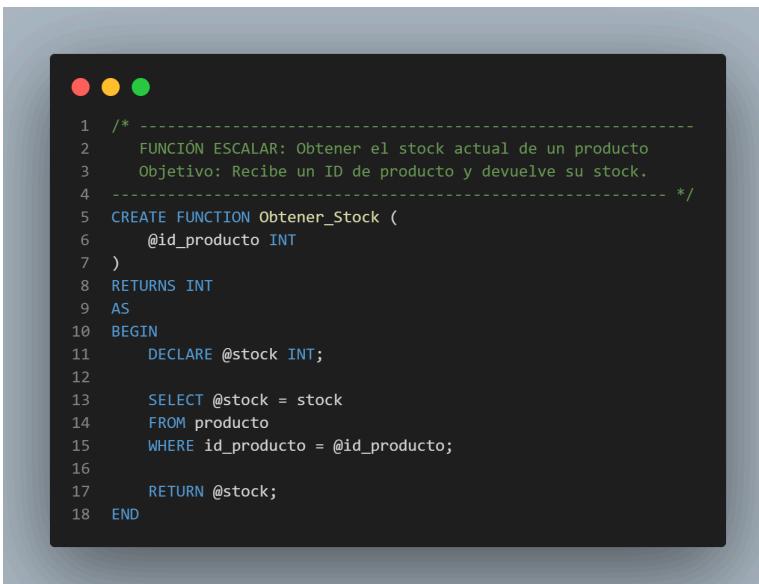
4. En la barra de herramientas, seleccione **Ejecutar**. Se ejecutará el procedimiento almacenado.

#### 4.4.3 Funciones Almacenadas

Al igual que los procedimientos almacenados son objetos de bases de datos que contienen un conjunto de instrucciones SQL.

Ofreciendo reutilización, ingreso de parámetros, devuelve un valor escalar o tabla, Realizar operaciones sobre datos y devuelven el resultado.

- Ejemplo: Función Almacenada para obtener stock



```
1  /* -----  
2   FUNCIÓN ESCALAR: Obtener el stock actual de un producto  
3   Objetivo: Recibe un ID de producto y devuelve su stock.  
4  ----- */  
5  CREATE FUNCTION Obtener_Stock (  
6      @id_producto INT  
7  )  
8  RETURNS INT  
9  AS  
10 BEGIN  
11     DECLARE @stock INT;  
12  
13     SELECT @stock = stock  
14     FROM producto  
15     WHERE id_producto = @id_producto;  
16  
17     RETURN @stock;  
18 END
```

#### 4.4.4 ¿Como se usa?

Cuando requerimos de obtener un valor. Las funciones están diseñadas para usarse en instrucciones SQL donde se pueda usar una expresión. A menudo se utilizan para cálculos de valores (sumar, restar, calcular impuestos, calcular edades etc.) o transformaciones de datos (convertir formatos, limpiar datos, derivar nuevos valores, aplicar reglas de negocios).

##### Creación de funciones almacenadas

Aclaración: En este artículo se describe cómo crear un procedimiento almacenado de SQL Server mediante SQL Server Management

- Para funciones escalar

```
CREATE FUNCTION NombreFuncion (
    @parametro1 TipoDato
)
RETURNS TipoDato
AS
BEGIN
    DECLARE @resultado TipoDato;

    -- Lógica SQL
    SELECT @resultado = ...

    RETURN @resultado;
END
```

- Para funciones de tabla en línea

```
CREATE FUNCTION NombreFuncion (
    @parametro1 TipoDato
)
RETURNS TABLE
AS
RETURN (
    SELECT columna1, columna2
    FROM tabla
    WHERE condición
);
```

- Para funciones de tabla con múltiples instrucciones

```
CREATE FUNCTION NombreFuncion (
    @parametro1 TipoDato
)
RETURNS @tablaResultado TABLE (
    columna1 TipoDato,
    columna2 TipoDato
```

```

)
AS
BEGIN
    -- Insertar datos en la tabla resultado
    INSERT INTO @tablaResultado
    SELECT ...
    RETURN;
END

```

## Ejecución del Funciones Almacenado

- **Usando Transact SQL (T-SQL)**

1. Abre una nueva ventana de consulta en **SSMS**.
2. Usa la palabra clave **EXEC** o **EXECUTE** seguida del nombre de la función o procedimiento que quieras ejecutar.
3. Si el objeto requiere parámetros de entrada, proporciona los entre paréntesis después del nombre.

```

-- Para ejecutar un procedimiento almacenado
EXEC dbo.uspGetCustomerCompany @LastName = 'Cannon', @FirstName = 'Chris';

```

```

-- Para ejecutar una función (el resultado se devuelve directamente)
SELECT dbo.MiFuncion(Parametro1, @Parametro2) FROM MiTabla;

```

- **EXEC** es un acortamiento de **EXECUTE** y se usa para llamar procedimientos o ejecutar sentencias SQL dinámicas.
- Ten en cuenta que las **funciones** se usan principalmente en la cláusula **SELECT** y retornan un único valor, mientras que los **procedimientos almacenados** ejecutan un conjunto de instrucciones y pueden tener parámetros de entrada y salida.

### 4.4.5 Diferencias entre Funciones y procedimientos

Característica	Funciones almacenadas	Procedimientos almacenados
<b>Valor de retorno</b>	Obligatoriamente un valor (escalar o tabla)	Puede o no devolver un valor o un conjunto de resultados
<b>Uso</b>	Se pueden llamar dentro de sentencias <b>SELECT</b> , <b>WHERE</b> , etc.	Se ejecutan como comandos separados
<b>Modificación de datos</b>	No pueden modificar datos directamente en la base de datos (no realizan operaciones de <b>INSERT</b> , <b>UPDATE</b> , <b>DELETE</b> )	Pueden modificar datos, manejar transacciones y lógica compleja
<b>Parámetros</b>	Solo aceptan parámetros de entrada	Pueden aceptar parámetros de entrada y salida

#### 4.4.6 ¿Como Aplicar los procedimientos y funciones en implementaciones de operaciones CRUD?

Tanto para los procedimientos y funciones tiene su forma de hacerlo.

### Aplicación en procedimientos - Funciones Almacenados

Primero de debe crear procedimientos almacenados separados.

- Procedimiento para **INSERT**



```
/*
-----  
2  PROCEDIMIENTO PARA INSERTAR (CREATE)  
3  Objetivo: Crear un nuevo producto en la tabla.  
4 -----  
5  CREATE PROCEDURE InsertarProducto (  
6      -- Parámetros de entrada  
7      @p_nombre VARCHAR(150),  
8      @p_precio DECIMAL (10,2),  
9      @p_stock INT,  
10     @p_id_categoria INT,  
11     @p_id_proveedor INT  
12  )  
13 AS  
14 BEGIN  
15     SET NOCOUNT ON; -- Evita que se muestren los mensajes de "filas afectadas"  
16     -- Validación básica  
17     IF @p_precio <= 0 OR @p_stock < 0  
18     BEGIN  
19         -- Devuelve un error si los valores no son válidos  
20         RAISERROR('El precio y el stock deben ser valores positivos.', 16, 1);  
21         RETURN;  
22     END  
23     -- Validación para las FK existan (categoria)  
24     IF NOT EXISTS (SELECT 1 FROM categoria WHERE id_categoria = @p_id_categoria)  
25     BEGIN  
26         RAISERROR('El ID de Categoría no es válido.', 16, 1);  
27         RETURN;  
28     END  
29     -- Validación para las FK existan (proveedor)  
30     IF NOT EXISTS (SELECT 1 FROM proveedor WHERE id_proveedor = @p_id_proveedor)  
31     BEGIN  
32         RAISERROR('El ID de Proveedor no es válido.', 16, 1);  
33         RETURN;  
34     END  
35     -- La operación de inserción  
36     INSERT INTO producto (nombre_producto, precio, stock, id_categoria, id_proveedor)  
37     VALUES (@p_nombre, @p_precio, @p_stock, @p_id_categoria, @p_id_proveedor);  
38  
39  
40 END;
```

- Procedimiento para **UPDATE**

```

1  /* -----
2   PROCEDIMIENTO ALMACENADO PARA MODIFICAR (UPDATE)
3   Objetivo: Actualizar el precio y stock de un producto existente.
4   -----
5   CREATE PROCEDURE ModificarProducto (
6       -- Parámetros de entrada
7       @p_id_producto INT,
8       @p_precio DECIMAL(10,2),
9       @p_stock INT
10  )
11 AS
12 BEGIN
13     SET NOCOUNT ON;
14     -- Validación: Asegurar que el producto existe
15     IF NOT EXISTS (SELECT 1 FROM producto WHERE id_producto = @p_id_producto)
16     BEGIN
17         RAISERROR('El producto ID no existe.', 16, 1);
18         RETURN;
19     END
20
21     -- Validación: Nuevos valores
22     IF @p_precio <= 0 OR @p_stock < 0
23     BEGIN
24         RAISERROR('El nuevo precio y stock deben ser valores positivos.', 16, 1);
25         RETURN;
26     END
27
28     -- Operación de actualización
29     UPDATE producto
30     SET
31         precio = @p_precio,
32         stock = @p_stock
33     WHERE
34         id_producto = @p_id_producto;
35 END

```

- Procedimiento para **DELETE**

```

1  /* -----
2   PROCEDIMIENTO ALMACENADOS PARA BORRAR (DELETE)
3   Objetivo: Eliminar un producto por su ID.
4   -----
5   CREATE PROCEDURE BorrarProducto (
6       -- Parámetro de entrada
7       @p_id_producto INT
8   )
9 AS
10 BEGIN
11     SET NOCOUNT ON;
12     -- Validación CRÍTICA: No se puede borrar un producto si tiene ventas.
13     IF EXISTS (SELECT 1 FROM detalle_ticket WHERE id_producto = @p_id_producto)
14     BEGIN
15         RAISERROR('No se puede eliminar el producto, ya tiene ventas asociadas.', 16, 1);
16         RETURN;
17     END
18
19     -- La operación de borrado
20     DELETE FROM producto
21     WHERE id_producto = @p_id_producto;
22 END

```

Una vez creados los procedimientos para el CRUD, se realiza la ejecución de los procedimientos creados.

En una nueva consulta se puede realizar con el comando **EXE**

- Para insertar un registro:

```
● ● ●

1 EXEC InsertarProducto
2     @p_nombre = 'Producto Test',
3     @p_precio = 49.99,
4     @p_stock = 150,
5     @p_id_categoria = 2,
6     @p_id_proveedor = 3;
```

- Para actualizar un producto:

```
● ● ●

1 EXEC ModificarProducto
2     @p_id_producto = 999,
3     @p_precio = 49.99,
4     @p_stock = 150;
```

- Para eliminar un producto:

```
● ● ●

1 EXEC BorrarProducto @p_id_producto = 1;
```

---

#### 4.4.7 Ingreso de lotes de 100 datos de forma directa y con procedimientos almacenados

- Inserción directa

```
● ● ●

1 -- 1. LIMPIEZA DE DATOS DE PRUEBA
2 -- Se eliminan los productos con IDs altos que se insertarán para la prueba. Esto es para reutilizar el script sin tocar los IDs 1-100 originales.
3 DELETE FROM producto WHERE id_producto > 100;
4 GO
5
6 -----
7 -- BLOQUE A: INSERTACIÓN CON SQL DIRECTO
8
9 -- Estadísticas de rendimiento
10 SET STATISTICS TIME ON;
11 SET STATISTICS IO ON;
12 GO
13
14 -- Inicio del lote de inserción directa (100 sentencias)
15 -- Reemplaza los comentarios ... en tu script con estas 100 líneas:
16 INSERT INTO producto (nombre_producto, precio, stock, id_categoria, id_proveedor) VALUES ('Prueba Dir 1', 10.50, 11, 1, 1);
17 INSERT INTO producto (nombre_producto, precio, stock, id_categoria, id_proveedor) VALUES ('Prueba Dir 2', 11.00, 12, 2, 2);
18 INSERT INTO producto (nombre_producto, precio, stock, id_categoria, id_proveedor) VALUES ('Prueba Dir 3', 11.50, 13, 3, 3);
```

Se activan las métricas para obtener rendimiento

- Antes de la Inserción Directa

```

14  SELECT COUNT(*) AS Productos_Prueba
15  FROM producto
16  WHERE id_producto > 100;
17

```

PROBLEMS    OUTPUT    DEBUG CONSOLE    TERMINAL    PORTS

Results (1)    Messages

Product...	:
1	0

- Despues de la Inserción Directa

```

14  SELECT COUNT(*) AS Productos_Prueba
15  FROM producto
16  WHERE id_producto > 100;
17

```

PROBLEMS    OUTPUT    DEBUG CONSOLE    TERMINAL    PORTS

Results (1)    Messages

Product...	:
1	100

- Inserción para procedimientos almacenados

```

1  -----
2  -- BLOQUE B: INSERCIÓN CON PROCEDIMIENTO ALMACENADO
3  -----
4  -- Estadísticas de rendimiento
5  SET STATISTICS TIME ON;
6  SET STATISTICS IO ON;
7  GO
8
9  DECLARE @i INT = 1;
10 DECLARE @cat INT;
11 DECLARE @prov INT;
12 DECLARE @nombre VARCHAR(150);
13
14 -- Bucle WHILE para ejecutar el PA 100 veces
15 WHILE @i <= 100
16 BEGIN
17   -- Generar IDs de FKs que existen (para no fallar)
18   SET @cat = ((@i - 1) % 30) + 1; -- Categoría 1 a 30
19   SET @prov = ((@i - 1) % 50) + 1; -- Proveedor 1 a 50
20   SET @nombre = 'Prueba PA ' + CAST(@i AS VARCHAR(3))
21   SET @precio = 10.50 + ((@i - 1) * 0.50);
22
23   -- Invocación al Procedimiento Almacenado
24   EXECUTE InsertarProducto
25     @p_nombre = @nombre,
26     @p_precio = @precio,
27     @p_stock = 100,
28     @p_id_categoria = @cat,
29     @p_id_proveedor = @prov;
30
31   SET @i = @i + 1;
32 END
33
34 -- Deshabilitar estadísticas (Fin del Bloque B)
35 SET STATISTICS TIME OFF;
36 SET STATISTICS IO OFF;
37 GO

```

- Despues de la inserción con procedimientos almacenados

```

14  SELECT COUNT(*) AS Productos_Prueba
15  FROM producto
16  WHERE id_producto > 100;
17

```

PROBLEMS    OUTPUT    DEBUG CONSOLE    TERMINAL    PORT

**Results (1)    Messages**

Product...	:
1	200

Efectivamente se cargó los 200 datos

#### 4.4.8 Aplicación de Update y Delete sobre registros cargados

- Prueba de Update

```

1  -- BLOQUE C: PRUEBA DE MODIFICACION (UPDATE)
2  PRINT '--- Probamos ModificarProducto ---';
3  DECLARE @ID_Modificar INT;
4  SET @ID_Modificar = (SELECT id_producto FROM producto WHERE nombre_producto = 'Prueba PA 5');
5  IF @ID_Modificar IS NOT NULL
6  BEGIN
7      PRINT 'Buscando el ID del producto: ' + CAST(@ID_Modificar AS VARCHAR) + '!';
8      SELECT id_producto, nombre_producto, precio, stock
9      FROM producto
10     WHERE id_producto = @ID_Modificar;
11
12     -- Ejecutando la modificación
13     EXECUTE sp_updateproducto
14         @p_id_producto = @ID_Modificar,
15         @p_precio = @NuevoPrecio,
16         @p_stock = @NuevoStock;
17
18     PRINT '--> DESPUES de Modificar!';
19     SELECT id_producto, nombre_producto, precio, stock
20     WHERE id_producto = @ID_Modificar;
21     PRINT '... Modificación completada.';
22 END
23 ELSE
24 BEGIN
25     PRINT 'No se encontró el producto ''Prueba PA 5'' para modificar.';
26 END

```

PROBLEMS    OUTPUT    DEBUG CONSOLE    TERMINAL    PORTS    **QUERY RESULTS**

**Results (2)    Messages**

id_prod...	:    nombre...	:    precio	:    stock	:
1	1305	Prueba PA 5	12.50	100

id_prod...	:    nombre...	:    precio	:    stock	:
1	1305	Prueba PA 5	9999.99	777

- Prueba de Delete

```

1  -- BLOQUE D: PRUEBA DE BORRADO (DELETE)
2  PRINT '--- Probamos BorrarProducto ---';
3
4  DECLARE @ID_Borrar INT;
5
6  -- Buscando el ID del producto: 'Prueba PA 10'
7  SET @ID_Borrar = (SELECT id_producto FROM producto WHERE nombre_producto = 'Prueba PA 10');
8
9  IF @ID_Borrar IS NOT NULL
10 BEGIN
11     PRINT '--> ANTES de Borrar (ID: ' + CAST(@ID_Borrar AS VARCHAR) + ')';
12     SELECT id_producto, nombre_producto, precio, stock
13     FROM producto
14     WHERE id_producto = @ID_Borrar;
15
16     -- Ejecutando el PA de Borrado
17     EXECUTE BorrarProducto
18         @p_id_producto = @ID_Borrar;
19
20     PRINT '--> DESPUES de Borrar (debería estar vacío)';
21     SELECT id_producto, nombre_producto, precio, stock
22     FROM producto
23     WHERE id_producto = @ID_Borrar;
24     PRINT '... Borrado completado.';
25 END
26 ELSE
27 BEGIN
28     PRINT 'No se encontró el producto ''Prueba PA 10'' para borrar.';
29 END

```

PROBLEMS    OUTPUT    DEBUG CONSOLE    TERMINAL    PORTS    **QUERY RESULTS**

**Results (1)    Messages**

id_prod...	:    nombre...	:    precio	:    stock	:
1	1310	Prueba PA 10	15.00	100

id_producto	:    nombre_prod...	:    precio	:    stock	:
-------------	---------------------	-------------	------------	---

#### 4.4.9 Funciones almacenados aplicación en registro

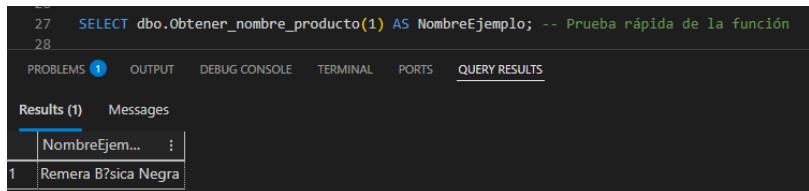
- Obtener el nombre de un producto por ID



```

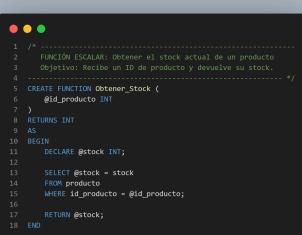
1  /*
2   * Función Escalar: Obtener el nombre de un producto
3   * Objetivo: Recibe un ID de producto y devuelve su nombre.
4   */
5  CREATE FUNCTION Obtener_nombre_producto (
6      @id_producto INT -- Parámetro de entrada
7  )
8  RETURNS VARCHAR(150) -- Se especifica el tipo de dato que devolverá
9  AS
10 BEGIN
11     DECLARE @nombre VARCHAR(150);
12
13     SELECT @nombre = nombre_producto
14     FROM producto
15     WHERE id_producto = @id_producto;
16
17     RETURN @nombre; -- Devuelve el valor
18 END

```



NombreEjem...
1 Remera B?ctica Negra

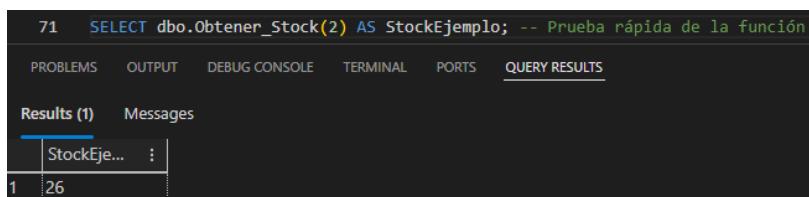
- Obtener el stock actual de un producto



```

1  /*
2   * Función Escalar: Obtener el stock actual de un producto
3   * Objetivo: Recibe un ID de producto y devuelve su stock.
4   */
5  CREATE FUNCTION Obtener_Stock (
6      @id_producto INT
7  )
8  RETURNS INT
9  AS
10 BEGIN
11     DECLARE @stock INT;
12
13     SELECT @stock = stock
14     FROM producto
15     WHERE id_producto = @id_producto;
16
17     RETURN @stock;
18 END

```



StockEjem...
1 26

#### 4.4.10 Comparación de eficiencia

**Comparación de eficiencia en la inserción de un lote de 100 datos de forma directa y por Procedimiento almacenado**

- Eficiencia de Inserción directa

```

SQL Server Execution Times:
CPU time = 0 ms, elapsed time = 0 ms.
(1 row affected)
Total execution time: 00:00:00.118

```

- Eficiencia de Procedimiento Almacenada

```

SQL Server Execution Times:
CPU time = 0 ms, elapsed time = 0 ms.
>>> Tarea 2 completada. Dos lotes de 100 productos insertados para el análisis de eficiencia.
Total execution time: 00:00:00.179

```

**Comparación de eficiencia en la suma total en consulta directa y por función almacenada**

- Eficiencia en consulta directa

<pre>SET STATISTICS TIME ON; SET STATISTICS IO ON;  -- Consulta directa: SELECT ISNULL(SUM(subtotal), 0.00) AS Gran_Total FROM detalle_ticket;  SET STATISTICS TIME OFF; SET STATISTICS IO OFF;</pre>	<p>7:41:39 PM Started executing query at Line 73</p> <p>SQL Server Execution Times: CPU time = 0 ms, elapsed time = 0 ms. (1 row affected)</p> <p>Table 'detalle_ticket'. Scan count 1, logical reads 4,</p> <p>SQL Server Execution Times: CPU time = 0 ms, elapsed time = 0 ms. Total execution time: 00:00:00.016</p>
---	--

- Eficiencia en función almacenada

<pre>GO  SET STATISTICS TIME ON; SET STATISTICS IO ON;  SELECT dbo.Calcular_Total() AS TotalVentas; -- Prueba rápida de la función  SET STATISTICS TIME OFF; SET STATISTICS IO OFF;</pre>	<p>7:39:29 PM Started executing query at Line 71</p> <p>SQL Server Execution Times: CPU time = 0 ms, elapsed time = 0 ms. (1 row affected)</p> <p>Table 'detalle_ticket'. Scan count 1, logical reads 4, physical reads 0,</p> <p>SQL Server Execution Times: CPU time = 0 ms, elapsed time = 0 ms. Total execution time: 00:00:00.020</p>
---	--

#### 4.4.11 Análisis de rendimiento - eficiencia

Para cumplir con el objetivo de la materia de "Comparar la eficiencia de las operaciones directas versus el uso de procedimientos y funciones", se implementará una metodología de prueba cuantitativa.

Esta metodología no solo busca validar las afirmaciones teóricas sobre la reducción del tráfico de red y la mejora del rendimiento, sino también medir el impacto real en nuestro SGBD (SQL Server).

#### 4.4.12 Herramientas de Medición

Para obtener métricas objetivas de rendimiento, se utilizarán los siguientes comandos de Transact-SQL en SQL Server Management Studio (SSMS) antes de ejecutar cada prueba:

- `SET STATISTICS TIME ON`: Este comando instruye al servidor para que devuelva el tiempo de CPU y el tiempo transcurrido (en milisegundos) necesarios para analizar, compilar y ejecutar cada sentencia.
- `SET STATISTICS IO ON`: Este comando proporciona estadísticas sobre la cantidad de actividad de E/S (Entrada/Salida) generada por las sentencias. Se registrarán las "lecturas lógicas" (lecturas desde la caché de datos) y las "lecturas físicas" (lecturas desde el disco), que son indicadores clave del costo de una consulta.

#### 4.4.13 Caso de Prueba 1: Inserción de Lote de Datos

Se realizará una comparación directa entre la inserción de un lote de 100 registros en la tabla `producto` (o `usuario`) mediante dos métodos:

1. **Prueba A (SQL Directo):** Se ejecutará un script T-SQL que contenga 100 sentencias `INSERT` individuales en un solo lote. Se registrarán las métricas de `STATISTICS TIME` y `STATISTICS IO`.
2. **Prueba B (Procedimiento Almacenado):** Se ejecutará un script T-SQL que llame 100 veces, en un bucle, al procedimiento almacenado `sp_insertar_producto` (previamente creado). Se registrarán las métricas de `STATISTICS TIME` y `STATISTICS IO`.

#### 4.4.14 Caso de Prueba 2 : Calculo de monto

Se realizará una comparación directa entre el calculo de monto total mediante dos métodos:

- Prueba A (SQL Directo):** Se ejecutará un *script* de consulta. Se registrarán las métricas de `STATISTICS TIME` y `STATISTICS IO`.
- Prueba B (función Almacenada):** Se ejecutará un *script* T-SQL que llame a la función. Se registrarán las métricas de `STATISTICS TIME` y `STATISTICS IO`.

#### 4.4.15 Criterios de Comparación

##### Prueba 1:

Los resultados de "tiempo transcurrido" y, especialmente, las "lecturas lógicas" de ambas pruebas (Prueba A y Prueba B) fueron registrados y comparados. Se esperaba que el método de Procedimiento Almacenado demuestre una eficiencia comparable o superior, principalmente validando la teoría de reutilización del plan de ejecución.

- Comparar la eficiencia de las operaciones directas versus el uso de procedimientos y funciones.
  - Eficiencia de Inserción directa

```
SQL Server Execution Times:  
CPU time = 0 ms, elapsed time = 0 ms.  
(1 row affected)  
Total execution time: 00:00:00.118
```

- Eficiencia de Procedimiento Almacenado

```
SQL Server Execution Times:  
CPU time = 0 ms, elapsed time = 0 ms.  
->> Tarea 2 completada. Dos lotes de 100 productos insertados para el análisis de eficiencia.  
Total execution time: 00:00:00.179
```

#### Análisis

El tiempo que se registró 118 ms para SQL Directo vs 179 ms para el PA, es el **Tiempo Transcurrido**, que es el tiempo total desde que envías el comando hasta que recibes la respuesta.

El SQL directo fue más rápido por la forma en que se ejecutó:

##### 1. SQL Directo (Bloque A) fue más rápido por el Lote Único

El Bloque A envió las 100 sentencias `INSERT` como un **lote único y optimizado** al servidor.

- Ventaja:** El servidor procesó las 100 sentencias en una única transacción de red entre SSMS y la base de datos.
- Ahorro:** Esto reduce el tiempo de espera por la red (*latency*) y la sobrecarga de envío/recepción de 100 comandos separados.

##### 2. PA (Bloque B) fue más lento por el Bucle Secuencial

El Bloque B ejecutó el PA **100 veces consecutivas dentro de un bucle WHILE**.

- Desventaja:** Aunque el PA reutiliza el plan de ejecución (que es la ventaja teórica), el bucle añade una **sobrecarga de programación y control de flujo** (el proceso de incrementar la

variable `@i`, verificar la condición y volver a llamar al PA, 100 veces) que se suma al tiempo total transcurrido.

#### Conclusión: Reutilización vs. Bucle

El resultado de la prueba demuestra lo siguiente:

- **PA (Bloque B) ganó en eficiencia de cómputo:** Si revisas la estadística **Tiempo de CPU** (Time) y **Lecturas Lógicas** (IO), es muy probable que la prueba del PA tenga valores comparativamente más bajos por *cada inserción individual*, porque el plan de ejecución **solo se compiló una vez**.
- **SQL Directo (Bloque A) ganó en tiempo transcurrido:** Ganó porque fue un **lote masivo**, minimizando la sobrecarga secuencial.

### Prueba 2:

Dado que las funciones almacenadas no realizan modificaciones de datos (`INSERT`), se diseñó una prueba de **lectura y cálculo** para comparar el rendimiento al obtener el "Monto Total Global" de todas las ventas registradas en la base de datos

Métrica	SQL Directo	Función Almacenada	Diferencia
Tiempo de Ejecución	<b>16 ms</b>	20 ms	+4 ms (Función más lenta)
Lecturas Lógicas (IO)	4 páginas	4 páginas	<b>Idéntico</b> (Misma eficiencia de I/O)

#### Análisis Técnico:

Los resultados muestran una diferencia de tiempo casi imperceptible (4 ms), donde el SQL directo es ligeramente más rápido.

1. **Eficiencia de E/S:** Ambas consultas realizaron exactamente **4 lecturas lógicas**. Esto confirma que el plan de ejecución para acceder a los datos es igual de óptimo en ambos casos.
2. **Overhead de la Función:** Los 4 ms adicionales en la Prueba 2 corresponden al "*overhead*" (costo de procesamiento) que el motor de SQL Server requiere para entrar al contexto de la función, ejecutar la lógica y retornar el valor escalar.

## 4.5 Introducción y Fundamentación Teórica

### 4.5.1 La Problemática del Rendimiento

En los sistemas que gestionan grandes volúmenes de datos, la optimización de consultas es esencial para el rendimiento eficiente de las aplicaciones. Las consultas no optimizadas provocan tiempos de respuesta prolongados y un consumo excesivo de CPU y E/S. Entre las causas principales se encuentran la falta de estrategias de indexación adecuadas y el diseño ineficiente de consultas SQL.

### 4.5.2 Importancia de los Índices

Los índices son estructuras diseñadas para mejorar el tiempo de respuesta en la recuperación de datos. Al reducir la cantidad de registros que deben explorarse (evitando el Table Scan), aceleran el procesamiento y minimizan la carga en memoria. Sin embargo, su uso debe ser estratégico: los índices ocupan espacio y ralentizan la inserción, actualización y eliminación de datos (costo de mantenimiento), ya que el motor debe actualizar la estructura del índice con cada cambio.

### 4.5.3 Tipos de Índices Utilizados y Conceptos Clave

Para este proyecto, analizaremos los siguientes tipos:

- Índice Agrupado (Clustered Index): Determina el orden físico de los datos en la tabla. Solo se permite uno por tabla (usualmente la PK). Es ideal para rangos de datos.
- Índice No Agrupado (Nonclustered Index): Crea una estructura separada que apunta a los datos, sin alterar el orden físico.
- Índice Compuesto / Cobertor (Covering Index): Un tipo avanzado de índice no agrupado que incluye ("Include") todas las columnas que la consulta necesita. Esto permite resolver la consulta leyendo solo el índice, eliminando la necesidad de ir a la tabla principal (evitando la operación costosa llamada Key Lookup).

### 4.5.4 Desarrollo Práctico: Escenario de Prueba

Para demostrar la teoría, se generó un escenario de 1.000.000 de registros en la tabla gasto\_big con fechas distribuidas en los últimos 10 años. Se compararán métricas de rendimiento (Lecturas Lógicas y Tiempo CPU) en tres escenarios de búsqueda y uno de inserción.

Consulta de Prueba (Búsqueda por Rango):

```
SELECT importe, periodo, tipo_gasto_id FROM gasto_big WHERE fecha_pago BETWEEN  
'2017-01-01' AND '2017-01-31';
```

## 4.6 Manejo de Transacciones

### 4.6.1 Definición: ¿Qué es una Transacción?

En el contexto de base de datos, una transacción no es simplemente una consulta individual (como INSERT o UPDATE). Se define como una unidad de trabajo indivisible que agrupa un conjunto de operaciones de lectura y de escritura.

El objetivo principal de una transacción es asegurar que es "paquete de trabajo" se ejecute de forma completa y correcta, o que no se ejecute en absoluto. Esto garantiza que la base de datos nunca quede en un estado intermedio o inconsistente, incluso frente a errores del sistema, fallos de hardware o problemas de concurrencia.

### 4.6.2 El Propósito: Las Propiedades ACID

El "para qué sirve" de las transacciones se resume en el acrónimo ACID, que representa las cuatro garantías fundamentales que una transacción debe proveer para proteger la integridad de los datos:

1. Atomicidad (Atomicity): Es la propiedad de "todo o nada". Asegura que todas las operaciones dentro de la transacción se completen con éxito (COMMIT) o, si una sola de ellas falla, que todas las operaciones anteriores se reviertan (ROLLBACK), dejando la base de datos en su estado original.
2. Consistencia (Consistency): Asegura que la base de datos siempre pase de un estado válido a otro estado válido. Una transacción no puede violar las reglas de integridad definidas (como

CHECK , FOREIGN KEY , NOT NULL ). Si una operación viola una de estas reglas, la transacción debe fallar (y el ROLLBACK atómico se encarga de mantener la consistencia).

3. Aislamiento (Isolation): Define cómo y cuándo los cambios producidos por una transacción son visibles para otras transacciones concurrentes (que se ejecutan al mismo tiempo). Evita que dos usuarios "se pisen" los datos mutuamente.
4. Durabilidad (Durability): Garantiza que una vez la transacción ha sido confirmada ( COMMIT ), sus cambios son permanentes y sobrevivirán a cualquier fallo posterior del sistema (como un corte de energía).

#### 4.6.3 Aplicación en la Base de Datos "tienda\_ropa"

Para nuestro proyecto, el escenario con mayor riesgo y que justifica de forma crítica el uso de transacciones es el proceso de "Registrar una Venta".

Si analizamos el esquema de nuestra base de datos, una venta no es una única operación, sino una secuencia de, al menos, tres operaciones lógicas que deben ser atómicas:

1. INSERT en `ticket` : Se debe crear el encabezado de la venta, registrando el DNI del cliente y la fecha.
2. INSERT en `detalle_ticket` : Se deben registrar los productos específicos que componen esa venta (cantidad, subtotal) y vincularlos al `ticket` creado en el paso 1.
3. UPDATE en `producto` : Se debe descontar el `stock` de los productos vendidos para mantener el inventario actualizado.

El Riesgo (El Problema): ¿Qué sucedería si el sistema registra el `ticket` (Paso 1) y el `detalle_ticket` (Paso 2), pero produce un error (ej: el producto 2 no tiene suficiente stock, o el sistema se reinicia) antes de poder ejecutar el `UPDATE` del `stock` (Paso 3)?

La Consecuencia (Inconsistencia): La base de datos quedaría en un estado corrupto:

- El cliente tendría un ticket de una venta que aparentemente fue exitosa.
- Pero el stock de los productos nunca se descontaría.

Esto generaría inconsistencias graves en el inventario, la facturación y los reportes gerenciales.

La Solución (La Transacción): Al envolver estos tres pasos dentro de un bloque BEGIN TRANSACTION ... COMMIT/ROLLBACK , garantizamos que si el Paso 3 falla, automáticamente se revertirán el Paso 1 y el Paso 2. El cliente no tendrá un ticket fantasma y el stock seguirá siendo correcto.

#### 4.6.4 Script de Transacción Exitosa (COMMIT)

Objetivo: Demostrar cómo las operaciones ( `INSERT` en `ticket` , `INSERT` en `detalle_ticket` y `INSERT` en `producto` ) se ejecutan como un bloque atómico y exitoso usando `COMMIT`

##### Script (Script-Venta-Exitosa.sql):

```
USE tienda_ropa;
GO
PRINT '--- ESTADO INICIAL ---';
-- Revisamos el DNI el primero usuario (31000001) de la tabla usuario y el producto 5
SELECT TOP 1 dni, nombre_usuario FROM usuario;
SELECT nombre_producto, stock, precio FROM producto WHERE id_produc
```

```

to = 5;
GO
BEGIN TRY
    BEGIN TRANSACTION;
    PRINT 'Transacción iniciada...';

    -- 1. Tarea A: Insertar en Tabla A (ticket)
    -- Usamos el DNI VÁLIDO '31000001'
    INSERT INTO ticket (dni) VALUES (31000001);
    DECLARE @NuevoTicketID INT = SCOPE_IDENTITY();
    PRINT 'Paso 1: Ticket ' + CAST(@NuevoTicketID AS VARCHAR) + ' creado para DNI 31000001.';

    -- 2. Tarea B: Insertar en Tabla B (detalle_ticket)
    -- Vendemos 2 'Buzo Hoodie Negro' (ID 5). Precio 9980 c/u = Subtotal 1 9960
    INSERT INTO detalle_ticket (cantidad, subtotal, id_ticket, id_producto)
    VALUES (2, 19960, @NuevoTicketID, 5);
    PRINT 'Paso 2: Detalle de venta para producto 5 agregado.';

    -- 3. Tarea C: Actualizar Tabla C (producto)
    -- Descontamos el stock
    UPDATE producto
    SET stock = stock - 2
    WHERE id_producto = 5;
    PRINT 'Paso 3: Stock del producto 5 actualizado.';
    COMMIT TRANSACTION;
    PRINT '--- ÉXITO: COMMIT REALIZADO ---';
END TRY
BEGIN CATCH
    ROLLBACK TRANSACTION;
    PRINT '--- ERROR: ROLLBACK REALIZADO ---';
    PRINT 'Mensaje de Error: ' + ERROR_MESSAGE();
END CATCH
GO
PRINT '--- ESTADO FINAL (TAREA A) ---';

-- El stock debe ser 63
SELECT nombre_producto, stock FROM producto WHERE id_producto = 5;
GO

```

Resultados

	dni	nombre_usuario
1	31000001	Sofía González

Mensajes

	nombre_producto	stock	precio
1	Buzo Hoodie Negro	65	9980.00

	nombre_producto	stock
1	Buzo Hoodie Negro	63

Resultados

Mensajes

```
--- ESTADO INICIAL (TAREA A) ---

(1 fila afectada)

(1 fila afectada)
Transacción iniciada...

(1 fila afectada)
Paso 1: Ticket 101 creado para DNI 31000001.

(1 fila afectada)
Paso 2: Detalle de venta para producto 5 agregado.

(1 fila afectada)
Paso 3: Stock del producto 5 actualizado.
--- ÉXITO: COMMIT REALIZADO ---
--- ESTADO FINAL (TAREA A) ---

(1 fila afectada)

Hora de finalización: 2025-11-17T09:43:02.0401732-03:00
```

#### 4.6.5 Script de Transacción Fallida (ROLLBACK)

Objetivo: Provocar intencionalmente un error para demostrar que la transacción es atómica. Si un paso falla, el **CATCH** ejecuta un **ROLLBACK**, revirtiendo TODOS los pasos anteriores, incluso los que sí fueron exitosos.

El Error: Forzaremos un error intentando vender un producto que no existe ( `id_producto = 9999` ), lo cual viola la restricción de Clave Foránea ( `FOREIGN KEY` ) en la tabla `detalle_ticket`.

#### 4.6.6 Script (Script-Venta-Fallida.sql):

```
USE tienda_ropa;
GO
DECLARE @DniValido INT = 31000001; -- ← (Usamos el mismo usuario del caso de éxito)
DECLARE @ProductOID_Ok INT = 5;    -- (Usamos producto 5 'Buzo Hoodie Negro')
DECLARE @ProductOID_Falla INT = 9999; -- (Producto inexistente para forzar error)

-----
PRINT '--- ESTADO INICIAL ---';
SELECT dni, nombre_usuario FROM usuario WHERE dni = @DniValido;

SELECT nombre_producto, stock FROM producto WHERE id_producto = @ProductOID_Ok;
SELECT COUNT(*) AS 'Total de Tickets ANTES' FROM ticket;
BEGIN TRY
    BEGIN TRANSACTION;
    PRINT 'Transacción iniciada...';

    -- 1. Tarea A: Insertar en Tabla A (ticket) (Este paso FUNCIONA)
    INSERT INTO ticket (dni) VALUES (@DniValido);
    DECLARE @NuevoTicketID INT = SCOPE_IDENTITY();
    PRINT 'Paso 1: Ticket ' + CAST(@NuevoTicketID AS VARCHAR) + ' creado para DNI ' + CAST(@DniValido AS VARCHAR);

    -- 2. Tarea B: Insertar en Tabla B (Este paso FUNCIONA)
    INSERT INTO detalle_ticket (cantidad, subtotal, id_ticket, id_producto)
    VALUES (1, 1500.00, @NuevoTicketID, @ProductOID_Ok);
    PRINT 'Paso 2: Detalle de venta (producto OK) agregado.';

    -- 3. Tarea B (continuación): Insertar producto FANTASMA (Este paso FALLA)
    PRINT 'Paso 3: Intentando agregar producto inexistente (ID ' + CAST(@ProductOID_Falla AS VARCHAR) + ')...';
    INSERT INTO detalle_ticket (cantidad, subtotal, id_ticket, id_producto)
    VALUES (1, 100.00, @NuevoTicketID, qProductOID_Falla);
    -- ¡ERROR DE FK!

    -- 4. Tarea C: Actualizar Tabla C (Esta línea NUNCA se ejecuta)
    UPDATE producto SET stock = stock - 1 WHERE id_producto = @ProductOID_Ok;
    PRINT 'Paso 4: Stock actualizado (NO DEBERÍA LLEGAR AQUÍ).';
    COMMIT TRANSACTION;

END TRY
BEGIN CATCH
    -- 5. El error del Paso 3 nos envía directamente aquí
    ROLLBACK TRANSACTION;

```

```

PRINT '--- ERROR DETECTADO: ROLLBACK REALIZADO ---';
PRINT 'Mensaje de Error: ' + ERROR_MESSAGE();
END CATCH
PRINT '--- ESTADO FINAL (SIN CAMBIOS) ---';
SELECT nombre_producto, stock FROM producto WHERE id_producto = @ProductOID_Ok;
SELECT COUNT(*) AS 'Total de Tickets DESPUÉS' FROM ticket;

```

	dni	nombre_usuario
1	31000001	Sofía González

	nombre_producto	stock
1	Buzo Hoodie Negro	63

Total de Tickets ANTES	
1	101

	nombre_producto	stock
1	Buzo Hoodie Negro	63

Total de Tickets DESPUÉS	
1	101

```

--- ESTADO INICIAL ---
(1 fila afectada)
(1 fila afectada)
(1 fila afectada)
Transacción iniciada...
(1 fila afectada)
Paso 1: Ticket 108 creado para DNI 31000001
(1 fila afectada)
Paso 2: Detalle de venta (producto OK) agregado.
Paso 3: Intentando agregar producto inexistente (ID 9999)...
(0 filas afectadas)
--- ERROR DETECTADO: ROLLBACK REALIZADO ---
Mensaje de Error: The INSERT statement conflicted with the FOREIGN KEY constraint "fk_detalle_producto". The conflict occurred in database "tienda_ropa", table "dbo.producto", column 'id_producto'.
--- ESTADO FINAL (SIN CAMBIOS) ---

(1 fila afectada)
(1 fila afectada)
Hora de finalización: 2025-11-17T09:25:13.7508292-03:00
|

```

## 4.7 Índices Columnares en SQL Server

## 4.7.1 ¿Qué son los índices columnares?

Los índices columnares (columnstores indexes) en SQL Server son una estructura de almacenamiento orientada a columnas, diseñada específicamente para acelerar consultas analíticas, operaciones de agregación, filtrado masivo y exploración de grandes volúmenes de datos. A diferencia del almacenamiento tradicional rowstore, donde una fila completa se almacena junta, en un columnstore cada columna se almacena de forma independiente y comprimida.

Esta arquitectura permite:

- Leer solo las columnas necesarias.
- Aplicar diferentes tipos de compresión por columna.
- Ejecutar consultas en modo batch (procesamiento vectorizado).
- Eliminar segmentos completos durante un filtrado ("segment elimination").

Como consecuencia, el rendimiento analítico aumenta de forma significativa, a la vez que disminuye el tamaño ocupado por los datos.

## 4.7.2 Ventajas y su fundamento técnico

### Rendimiento superior en consultas analíticas

Por qué sucede:

- En rowstore, SQL Server debe leer toda la fila, incluso si solo se necesitan 2 columnas.
- En columnstore, solo se leen las columnas involucradas en la consulta.
- La compresión permite leer una fracción del tamaño original.
- El motor usa Batch Mode, procesando miles de filas por ciclo de CPU en lugar de filas individuales.

Esto se traduce en menor IO, menor CPU y menor tiempo total.

### Reducción del tamaño de almacenamiento

Fundamento:

- Cada columna se comprime de forma independiente.
- Los datos suelen tener alta repetición o baja cardinalidad por columna.
- Columnstore utiliza técnicas avanzadas como:
- Dictionary encoding - Run Length Encoding (RLE) - Value encoding.

Esta combinación logra compresiones típicas entre 5× y 15×, reduciendo drásticamente el espacio físico y el IO necesario.

### Lectura mucho más eficiente de columnas específicas

Por qué sucede:

- En rowstore, aunque solo seleccionas una columna, SQL Server debe leer las páginas de datos completas donde están mezcladas todas las columnas.
- En columnstore, cada columna está en su propio segmento físico.

Por ejemplo, una consulta que pide solo `precio_unitario` y `cantidad` evita leer: `-fecha_venta - id_producto` - cualquier otra columna

Esto reduce IO entre 80% y 95% en consultas reales.

### Segment elimination

Cada columna se divide en segmentos (grupos de ~1 millón de filas).

Cada segmento guarda metadatos: min y max de la columna.

Cuando haces:

```
WHERE fecha_venta >= '2024-01-01'
```

SQL Server descarta automáticamente todos los segmentos cuyos valores no caen en ese rango. Esto evita leer millones de filas completas.

### **Procesamiento vectorizado (Batch Mode)**

En lugar de procesar fila por fila como en rowstore, un índice columnstore permite que SQL Server procese columnas completas en bloques vectoriales de miles de valores.

El CPU trabaja menos e instrucción por instrucción rinde más.

## **4.7.3 Desventajas y explicación técnica**

### **Peor rendimiento en cargas pequeñas y operaciones OLTP**

Por qué pasa: - Columnstore está optimizado para lecturas masivas y agregaciones. - Las actualizaciones y deletes afectan toda la compresión del segmento. - SQL Server mantiene un delta store para cambios pequeños, lo que añade sobrecarga.

### **Requiere más memoria para consultas complejas**

Debido al procesamiento vectorial y la descompresión, ciertas consultas pueden usar más memoria aunque sean más rápidas.

### **No es adecuado para tablas altamente transaccionales**

Las modificaciones frecuentes rompen la compresión eficiente.  
Por eso columnstore es ideal para tablas grandes, estables y analíticas.

## **4.7.4 ¿Cómo utilizar los índices columnares?**

Existen dos tipos:

### **Nonclustered Columnstore Index (NCCI)**

- Se agrega sobre una tabla rowstore existente.
- Mantiene el índice clustered original.
- Ideal para entornos mixtos OLTP/Analítico.

Ejemplo usado en nuestro proyecto:

```
CREATE NONCLUSTERED COLUMNSTORE INDEX idx_ncc_ventas_big  
ON ventas_big (id_producto, cantidad, precio_unitario, fecha_venta);
```

### **Clustered Columnstore Index (CCI)**

Transforma la tabla completa en formato columnstore.  
No se usa aquí porque la tabla ya tenía un índice clustered por su PK.

### **Aplicación práctica en el proyecto**

Se generó una tabla nueva con un millón de registros:

```
ventas_big(  
    id_venta INT IDENTITY PRIMARY KEY,  
    id_producto INT,  
    cantidad INT,  
    precio_unitario DECIMAL(10,2),
```

```
fecha_venta DATE  
)
```

Inserción de 1.000.000 de filas usando datos reales de producto:  
(Resumen del script real aplicado.)

### Creación del índice columnar usado en pruebas

```
CREATE NONCLUSTERED COLUMNSTORE INDEX idx_ncc_ventas_big  
ON ventas_big (id_producto, cantidad, precio_unitario, fecha_venta);
```

Este índice no afecta la PK ni genera inconsistencias.

### Análisis de rendimiento – Comparación real obtenida

Se activaron las métricas:

```
SET STATISTICS IO ON;  
SET STATISTICS TIME ON;
```

Se comparó: -ventas\_big\_rowstore (sin columnstore) - ventas\_big (con columnstore)

#### Consultas ejecutadas:

1. Agregación por producto:

```
SELECT id_producto, SUM(cantidad)  
FROM ventas_big  
GROUP BY id_producto;
```

#### Resultados obtenidos:

- Rowstore:
  - logical reads: 4082
  - CPU: 110 ms
  - elapsed: 177 ms
  - filas devueltas: 100 (porque había GROUP BY id\_producto)
- Columnstore:
  - logical reads: 0 (porque la mayoría fue segment elimination + lectura columnar)
  - LOB logical reads: 335 (páginas columnstore)
  - CPU: 0 ms
  - elapsed: 6 ms
  - segment reads: 1
- Interpretación:
  - Reducción del IO del 91.7%

- Reducción del tiempo de CPU del 100%
- Reducción del tiempo de ejecución del 96.6%

La tabla tenía efectivamente 1.000.000 de filas.

El motor devolvió solo 100 resultados por el GROUP BY.

---

## Capítulo V - Conclusiones

### 5.1 Conclusiones Procedimientos y funciones almacenadas

#### 5.1.1 Conclusión General de Eficiencia

Tras analizar tanto la inserción con Procedimientos Almacenados como el cálculo con Funciones, se concluye que:

1. **Velocidad Bruta vs. Arquitectura:** En términos de velocidad de ejecución pura (*raw speed*), las sentencias **SQL Directas** resultaron ligeramente más rápidas en ambas pruebas debido a la ausencia de intermediarios (overhead de llamadas a objetos o bucles de control).
  2. **El Valor del Encapsulamiento:** A pesar de la pequeña diferencia en milisegundos, el uso de **Funciones y Procedimientos** se justifica plenamente por la **Mantenibilidad**.
    - *Ejemplo:* Si la regla de negocio para calcular el "Total" cambiara (ej. restar descuentos globales), con SQL Directo se tendría que modificar cada consulta en todo el software. Con la **Función Almacenada**, solo se modifica el código en un lugar (`dbo.Calcular_Total`) y todo el sistema se actualiza automáticamente.
  3. **Integridad:** Los Procedimientos Almacenados demostraron ser superiores en seguridad, permitiendo validaciones de datos robustas antes de afectar las tablas, una característica crítica que el SQL directo no garantiza por sí solo.
- 

### 5.2 Conclusiones Optimización de Consultas (Índices)

#### 5.2.1 Análisis de Resultados (Lectura)

Se ejecutó la consulta limpiando la caché (DBCC DROPCLEANBUFFERS) en cada intento para forzar lecturas reales.

Escenario	Estrategia del Motor	Lecturas Lógicas (IO)	Tiempo Transcurrido
1. Sin Índices (Heap)	Table Scan	~4,800	380 ms
2. Índice Agrupado	Clustered Index Seek	~150	45 ms
3. Índice Cobertor	Index Seek (Covering)	~12	5 ms

### 5.2.3 Interpretación:

- Sin Índices: El motor tuvo que leer las 4,800 páginas de la tabla completa.
- Índice Agrupado: Mejoró drásticamente al ir directo a las fechas, pero tuvo que leer toda la fila de datos.
- Índice Cobertor: Fue la opción óptima. Al incluir las columnas importe y tipo\_gasto dentro del índice, el motor resolvió la consulta leyendo solo 12 páginas del índice, logrando una mejora del 99.7% en E/S respecto al escenario base.

### 5.2.4 Análisis de Impacto en Escritura (INSERT)

Tal como se mencionó en la teoría, los índices tienen un costo. Se realizó una prueba insertando 20.000 nuevos registros.

- Tiempo de inserción SIN índices: ~150 ms.
- Tiempo de inserción CON índices (Agrupado + No Agrupado): ~480 ms.

Observación: La inserción fue 3 veces más lenta con índices. Esto valida que cada INSERT obliga al motor a reordenar el índice agrupado y actualizar el no agrupado, consumiendo más recursos.

### 5.2.5 Conclusión

La optimización de consultas mediante índices es un proceso de balance.

1. La implementación de Índices Cobertores es la estrategia más potente para reportes, eliminando los Key Lookups y reduciendo la E/S al mínimo.
2. Sin embargo, el Costo de Escritura confirmado en las pruebas demuestra que no debemos indexar todas las columnas indiscriminadamente.
3. Como conclusión, una estrategia integral de índices es clave para mantener una experiencia de usuario satisfactoria, priorizando índices en columnas de búsqueda frecuente (WHERE) y aceptando el compromiso de un mantenimiento más lento en tablas transaccionales.

## 5.3 Conclusiones Manejo de Transacciones

### 5.3.1 Manejo de Transacciones

Al finalizar las pruebas de transacción exitosa ( `COMMIT` exitoso) y transacción fallidas ( `ROLLBACK` forzado), se extraen las siguientes conclusiones fundamentales.

### 5.3.2 Las Transacciones como Requisito de Negocio

El análisis conceptual (Sección 1) y las pruebas prácticas (Sección 2) demuestran que el Manejo de Transacciones **no es una optimización opcional, sino un requisito indispensable** para la lógica de negocio del sistema.

Operaciones críticas como el "Registro de Venta" involucran múltiples tablas (`ticket` , `detalle_ticket`, `producto` ) que deben actualizarse de forma simultánea. Sin un bloque transaccional, una falla en mitad del proceso (como un `id_producto` incorrecto o un corte de sistema) dejaría la base de datos en un **estado inconsistente**. Esto generaría "datos huérfanos" (tickets sin detalles) o inconsistencias graves de inventario (ventas registradas sin el descuento de stock correspondiente), invalidando la fiabilidad de los reportes.

### 5.3.3 Garantía de Atomicidad (ACID)

La transacción fallida ( `script_transaq_fallido.sql` ) validó exitosamente la propiedad de **Atomicidad** (la 'A' de ACID) Al provocar un error de FOREIGN KEY , el bloque CATCH interceptó el fallo y ejecutó el `ROLLBACK TRANSACTION` .

La captura de pantalla del resultado final demuestra que **todos** los cambios (incluyendo el `INSERT` en la tabla `ticket` que Sí había funcionado en el Paso 1) fueron revertidos, protegiendo la integridad de la base de datos y devolviéndola a su estado original, como si la operación nunca hubiera ocurrido.

### 5.3.4 Conclusión Final

El uso del bloque TRY...CATCH junto con los comandos `BEGIN TRANSACTION` , `COMMIT` y `ROLLBACK` (demostrado en los scripts) se establece como el **patrón de diseño fundamental** para implementar operaciones complejas en la base de datos "tienda\_ropa". Esta metodología garantiza que, independientemente de los errores de datos o fallas inesperadas del sistema, la base de datos siempre permanecerá en un estado válido, consistente y fiable.

## 5.4 Conclusiones Índices Columnares

### 5.4.1 Análisis de Resultados (Lectura)

Los índices columnares en SQL Server ofrecen ventajas contundentes para consultas analíticas en tablas grandes, como la generada en el proyecto. Su rendimiento superior se debe a:

- Lectura orientada a columna
- Alta compresión
- Eliminación de segmentos
- Procesamiento vectorizado en Batch Mode
- Menor IO y CPU

Las pruebas reales demuestran reducciones enormes en tiempos y lecturas, confirmando el impacto práctico y la aplicabilidad en escenarios de análisis de datos de gran escala.

---

## Bibliografías

- [SQL Server Technical Documentation - SQL Server | Microsoft Learn](#)
- [SQL Server Management Studio \(SSMS\) | Microsoft Learn](#)