## **Variables y Asignación**

| -- Declarar una variable y asignarle un valor DECLARE @Precio DECIMAL(10,2); SET @Precio = 99.99;  -- Usar SELECT ... INTO para crear una nueva tabla a partir de una consulta SELECT EmployeeID, Name, Salary INTO EmpleadosTemp FROM Employees WHERE Salary > @Precio; |
| --- |

## **INSERTAR**

I

| NSERT INTO empleados (nombre, edad, departamento) VALUES ('Juan Pérez', 30, 'Ventas'); |
| --- |

## **Funciones de Manipulación de Texto**

| -- SUBSTR: Extrae una subcadena (ejemplo: primeros 4 caracteres) SELECT SUBSTR(Name, 1, 4) AS NombreCorto FROM Employees;  -- CONCAT: Une dos o más cadenas SELECT CONCAT(Name, ' - ', DepartmentID) AS InfoEmpleado FROM Employees;  -- LENGTH: Devuelve la longitud de una cadena SELECT Name, LENGTH(Name) AS LongitudNombre FROM Employees; |
| --- |

## **Funciones Matemáticas**

| -- ROUND: Redondea un número a la cantidad de decimales indicada SELECT Salary, ROUND(Salary, 0) AS SalarioRedondeado FROM Employees;  -- CEILING: Redondea hacia arriba SELECT Salary, CEILING(Salary) AS SalarioTecho FROM Employees;  -- FLOOR: Redondea hacia abajo SELECT Salary, FLOOR(Salary) AS SalarioPiso FROM Employees; |
| --- |

## **Cláusulas y Consultas Básicas**

| -- SELECT con WHERE, DISTINCT, LIKE, WHERE NOT, WHERE IN, condiciones con AND y BETWEEN  -- Seleccionar empleados con salario entre dos valores y cuyo nombre empiece con 'A' SELECT DISTINCT Name, Salary FROM Employees |
| --- |

| WHERE Name LIKE 'A%'   AND Salary BETWEEN 50000 AND 100000;  -- Excluir departamentos específicos SELECT Name, DepartmentID FROM Employees WHERE DepartmentID NOT IN (2, 5);  -- Combinar condiciones usando AND SELECT Name, Salary FROM Employees WHERE Salary > 60000 AND DepartmentID = 3; |
| --- |

## **Agrupación y Ordenación**

| -- Agrupar por DepartmentID y usar HAVING para filtrar grupos SELECT DepartmentID, COUNT(\*) AS TotalEmpleados, AVG(Salary) AS PromedioSalario FROM Employees GROUP BY DepartmentID HAVING COUNT(\*) > 5 ORDER BY PromedioSalario DESC; |
| --- |

## **Operadores y Funciones de Agregación**

| -- Uso de funciones de agregación y la cláusula CASE para clasificar SELECT   DepartmentID,  COUNT(\*) AS TotalEmpleados,  SUM(Salary) AS TotalSalarios,  AVG(Salary) AS PromedioSalario,  CASE   WHEN AVG(Salary) >= 80000 THEN 'Alto'  WHEN AVG(Salary) BETWEEN 50000 AND 79999 THEN 'Medio'  ELSE 'Bajo'  END AS CategoriaSalario FROM Employees GROUP BY DepartmentID; |
| --- |

## **Consultas Avanzadas: Subconsultas**

| -- Subconsulta para obtener empleados del departamento con el salario promedio más alto SELECT Name, Salary FROM Employees WHERE DepartmentID = (  SELECT TOP 1 DepartmentID  FROM Employees  GROUP BY DepartmentID  ORDER BY AVG(Salary) DESC ); |
| --- |

## **Consultas Avanzadas: CTE (Common Table**

| Expressions) -- CTE para calcular estadísticas por departamento y luego filtrar WITH DeptStats AS (  SELECT DepartmentID, COUNT(\*) AS TotalEmpleados, AVG(Salary) AS PromedioSalario  FROM Employees  GROUP BY DepartmentID ) SELECT DepartmentID, TotalEmpleados, PromedioSalario FROM DeptStats WHERE TotalEmpleados > 5; |
| --- |

## **Funciones de Ventana y Agregación**

| -- Seleccionamos los datos de los empleados SELECT   EmployeeID, -- ID del empleado  Name, -- Nombre del empleado  Salary, -- Salario del empleado  -- ROW\_NUMBER genera un número único de fila dentro de cada partición de departamento,   -- ordenando de mayor a menor salario  ROW\_NUMBER() OVER (PARTITION BY DepartmentID ORDER BY Salary DESC) AS NumFila,     -- RANK genera un rango basado en el orden de salario,   -- pero puede haber "saltos" si hay empleados con el mismo salario  RANK() OVER (PARTITION BY DepartmentID ORDER BY Salary DESC) AS RangoSalario,     -- DENSE\_RANK también genera un rango, pero sin saltos. Si hay empleados con el mismo salario,   -- ambos recibirán el mismo rango, pero no se salta el siguiente número.  DENSE\_RANK() OVER (PARTITION BY DepartmentID ORDER BY Salary DESC) AS RangoDenso  FROM Employees; -- De la tabla Employees |
| --- |

| -- Sintaxis de ROW\_NUMBER()  ROW\_NUMBER() OVER (PARTITION BY <columnas de partición> ORDER BY <columnas de orden>)   -- Sintaxis de RANK() RANK() OVER (PARTITION BY <columnas de partición> ORDER BY <columnas de orden>)  -- Sintaxis de DENSE\_RANK() DENSE\_RANK() OVER (PARTITION BY <columnas de partición> ORDER BY <columnas de orden>) |
| --- |

| -- Ejemplo de OVER con función de agregación (SUM) para calcular el total salarial por departamento SELECT   EmployeeID, -- ID del empleado  Name, -- Nombre del empleado  Salary, -- Salario del empleado  -- SUM calcula la suma total de salario para cada departamento (PARTITION BY DepartmentID),  -- pero no agrupa las filas. Así se puede ver el total de cada departamento junto con el salario individual.  SUM(Salary) OVER (PARTITION BY DepartmentID) AS TotalDepto  FROM Employees; -- De la tabla Employees |
| --- |

| SELECT    CONCAT(e.nombre, e.apellido) AS NombreEmpleado,  a.horas\_asignadas,   RANK() OVER (PARTITION BY e.depto\_id ORDER BY a.horas\_asignadas DESC) AS Mejores\_Rendimientos,   -- Ranking por departamento basado en horas asignadas  SUM(a.horas\_asignadas) OVER (PARTITION BY e.empleado\_id) AS MasHorasTrabajadas   -- Total de horas trabajadas por empleado  FROM   Empleados e  INNER JOIN   AsignacionesDeProyectos a ON a.empleado\_id = e.empleado\_id  INNER JOIN   Departamentos d ON e.depto\_id = d.depto\_id |
| --- |

**Nota:** Algunos motores (como MySQL) usan REGEXP para expresiones regulares, por ejemplo:

SELECT Name

FROM Employees

WHERE Name REGEXP '^[A-Z]';

## 

## **Otros Conceptos Clave**

### **Normalización de Bases de Datos (Conceptual)**

| -- 1NF: Cada columna contiene un valor atómico. -- 2NF: La tabla está en 1NF y cada columna depende de la clave primaria completa. -- 3NF: La tabla está en 2NF y no existen dependencias transitivas. |
| --- |

### **Procedimientos Almacenados**

| -- Crear un procedimiento almacenado para obtener un empleado por ID CREATE PROCEDURE GetEmployeeByID   @EmpID INT AS BEGIN  SELECT EmployeeID, Name, Salary, DepartmentID  FROM Employees  WHERE EmployeeID = @EmpID; END; |
| --- |

| CREATE PROCEDURE sp\_BuscarLibrosPorAutor   @NombreAutorParametro VARCHAR(100) -- Parámetro para buscar al autor AS BEGIN  SET NOCOUNT ON; -- Evita mensajes adicionales que interfieran con los resultados   -- Selecciona los títulos y autores de los libros  SELECT L.Titulo, L.Autor  FROM Libros L  -- Se utiliza LIKE para permitir coincidencias parciales en el nombre del autor  WHERE L.Autor LIKE '%' + @NombreAutorParametro + '%' ;     -- Fin del procedimiento END; |
| --- |

### **Índices**

| -- Crear un índice para acelerar la búsqueda por nombre CREATE INDEX idx\_EmployeeName ON Employees(Name); |
| --- |

### **Transacciones**

| -- Ejemplo de transacción con SAVEPOINT, COMMIT y ROLLBACK BEGIN TRANSACTION;   UPDATE Accounts  SET Balance = Balance - 100  WHERE AccountID = 1;   SAVE TRANSACTION SavePoint1;   UPDATE Accounts  SET Balance = Balance + 100  WHERE AccountID = 2;   -- Si ocurre un error, se puede hacer rollback a SavePoint1:  -- ROLLBACK TRANSACTION SavePoint1;  COMMIT; |
| --- |

### **Manejo de Errores**

| -- Señalizar un error si no hay suficiente stock IF (SELECT Stock FROM Products WHERE ProductID = 1) < 10 BEGIN  SIGNAL SQLSTATE '45000' SET MESSAGE\_TEXT = 'No hay suficiente stock'; END; |
| --- |

| IF condición BEGIN  -- Bloque de código si la condición es VERDADERA END ELSE BEGIN  -- Bloque de código si la condición es FALSA END |
| --- |

### **SP + TRANSACCIONES**

| CREATE PROCEDURE NombreDelProcedimiento  @Parametro1 INT, -- Parámetro de entrada, tipo entero (INT)  @Parametro2 NVARCHAR(50) -- Parámetro de entrada, tipo texto con un máximo de 50 caracteres (NVARCHAR) AS BEGIN  -- Inicia el bloque de código del procedimiento almacenado  BEGIN TRY  -- Comienza una transacción que agrupa las operaciones  BEGIN TRANSACTION;   -- Comienza la lógica de decisión utilizando la estructura IF  IF @Parametro1 > 100 -- Si el valor de @Parametro1 es mayor que 100  BEGIN  -- Si la condición es verdadera, se ejecuta el bloque siguiente  UPDATE Tabla1  SET columna1 = @Parametro2 -- Actualiza columna1 con el valor de @Parametro2  WHERE columna2 = @Parametro1; -- Aplica la actualización solo donde columna2 sea igual a @Parametro1  END  ELSE  BEGIN  -- Si la condición no es verdadera (es decir, @Parametro1 <= 100), se ejecuta este bloque  INSERT INTO Tabla2 (columnaA, columnaB) -- Inserta una nueva fila en Tabla2  VALUES (@Parametro1, @Parametro2); -- Coloca los valores de @Parametro1 y @Parametro2 en las columnas columnaA y columnaB  END   -- Si todas las operaciones anteriores fueron exitosas, se confirma la transacción  COMMIT TRANSACTION; -- Confirma los cambios realizados durante la transacción   END TRY  BEGIN CATCH  -- En caso de que ocurra un error, se ejecuta el bloque CATCH  -- Deshace los cambios realizados en la transacción si hubo algún error  ROLLBACK TRANSACTION; -- Revierte todos los cambios de la transacción   -- Declaración de variables para manejar el error  DECLARE @ErrorMessage NVARCHAR(4000), -- Variable para almacenar el mensaje de error  @ErrorSeverity INT, -- Variable para almacenar la gravedad del error  @ErrorState INT; -- Variable para almacenar el estado del error   -- Recupera los detalles del error generado  SELECT @ErrorMessage = ERROR\_MESSAGE(), -- Obtiene el mensaje de error  @ErrorSeverity = ERROR\_SEVERITY(), -- Obtiene la gravedad del error  @ErrorState = ERROR\_STATE(); -- Obtiene el estado del error   -- Lanza el error con la información recuperada para ser gestionado o visualizado  RAISERROR(@ErrorMessage, @ErrorSeverity, @ErrorState); -- Genera un error y lo lanza   END CATCH END; |
| --- |

**Verifique si hay suficiente stock** de un producto para poder realizar un pedido.

**Si hay suficiente stock**, se **crea el pedido**.

**Si no hay suficiente stock**, se lanza un error.

| CREATE PROCEDURE RealizarPedido  @ProductoID INT, -- El ID del producto que el cliente quiere comprar  @Cantidad INT, -- La cantidad de ese producto que el cliente quiere comprar  @ClienteID INT -- El ID del cliente que realiza el pedido AS BEGIN  -- Inicia el bloque TRY, que intentará realizar las operaciones  BEGIN TRY  -- Comienza una transacción, para asegurarnos de que todo se haga de forma atómica  BEGIN TRANSACTION; -- Comienza la transacción   -- Verificamos si hay suficiente stock disponible para el producto  DECLARE @StockDisponible INT;   SELECT @StockDisponible = Stock -- Traemos el stock disponible del producto  FROM Productos  WHERE ProductoID = @ProductoID; -- Filtramos por el ID del producto   -- Comprobamos si hay suficiente stock  IF @StockDisponible >= @Cantidad  BEGIN  -- Si hay suficiente stock, realizamos el pedido   -- Insertamos el nuevo pedido en la tabla Pedidos  INSERT INTO Pedidos (ClienteID, ProductoID, Cantidad)  VALUES (@ClienteID, @ProductoID, @Cantidad); -- Insertamos los valores del pedido   -- Actualizamos el stock del producto  UPDATE Productos  SET Stock = Stock - @Cantidad  WHERE ProductoID = @ProductoID; -- Descontamos la cantidad comprada del stock   -- Si todo sale bien, confirmamos la transacción  COMMIT TRANSACTION; -- Aquí confirmamos la transacción  END -- Fin del bloque IF (si hay suficiente stock)  ELSE  BEGIN  -- Si no hay suficiente stock, lanzamos un error  RAISERROR('No hay suficiente stock para completar el pedido.', 16, 1);  END -- Fin del bloque ELSE (si no hay suficiente stock)   END TRY -- Fin del bloque TRY (intento de ejecutar el código)   -- En caso de error, se maneja en el bloque CATCH  BEGIN CATCH  -- Si ocurrió algún error, deshacemos la transacción  ROLLBACK TRANSACTION; -- Aquí revertimos los cambios hechos hasta el momento   -- Recuperamos el mensaje de error  DECLARE @ErrorMessage NVARCHAR(4000), @ErrorSeverity INT, @ErrorState INT;  SELECT @ErrorMessage = ERROR\_MESSAGE(),  @ErrorSeverity = ERROR\_SEVERITY(),  @ErrorState = ERROR\_STATE();   -- Lanza el error para que lo vea el usuario  RAISERROR(@ErrorMessage, @ErrorSeverity, @ErrorState);  END CATCH -- Fin del bloque CATCH (manejo de errores)  END; -- Fin del procedimiento |
| --- |

### **Otros Objetos en SQL**

#### **Vistas**

| -- Crear una vista para mostrar información básica de empleados CREATE VIEW vw\_EmpleadosBasicos AS SELECT EmployeeID, Name, DepartmentID FROM Employees; |
| --- |

#### **Triggers**

| -- Crear un trigger que se ejecute después de insertar un nuevo empleado CREATE TRIGGER trg\_AfterInsertEmployee ON Employees AFTER INSERT AS BEGIN  PRINT 'Se ha insertado un nuevo empleado.'; END; |
| --- |

## **Funciones de Fechas (SQL Server)**

| -- Obtener la fecha y hora actual SELECT GETDATE() AS FechaActual;  -- Extraer partes de la fecha: Año, Mes, Día SELECT YEAR(GETDATE()) AS AñoActual,  MONTH(GETDATE()) AS MesActual,  DAY(GETDATE()) AS DiaActual;  -- Convertir una fecha a un formato específico SELECT CONVERT(DATE, GETDATE(), 1) AS FechaFormateada;  -- Sumar 5 días a la fecha actual SELECT DATEADD(DAY, 5, GETDATE()) AS FechaMas5Dias;  -- Calcular la diferencia en días entre dos fechas SELECT DATEDIFF(DAY, '2025-01-01', GETDATE()) AS DiferenciaDias;  -- Obtener el nombre del mes y del día de la semana SELECT DATENAME(MONTH, GETDATE()) AS NombreMes,  DATENAME(WEEKDAY, GETDATE()) AS NombreDiaSemana;  -- Extraer una parte específica de la fecha (por ejemplo, la hora) SELECT DATEPART(HOUR, GETDATE()) AS HoraActual; |
| --- |