GUSTAVO CORONEL
DESARROLLA SOFTWARE

PROGRAMACIÓN CON

TRANSACT-SQL

Eric Gustavo Coronel Castillo
https://gcoronelc.github.io
gcoronelc@gmail.com

LIMA – PERU - 2021

PROGRAMACIÓN CON TRANSACT SQL

Primera Edición

LIMA - PERÚ

PRESENTACIÓN

TRANSACT-SQL (T-SQL) extiende el estándar de SQL para incluir programación procedimental, funciones de usuario, variables locales, estructuras de control, control de errores, gestión de transacciones, etc.

T-SQL es un lenguaje muy potente que te permite definir casi cualquier tarea que quieras ejecutar sobre la base de datos; incluye características propias de cualquier lenguaje de programación, características que te permiten definir la lógica necesaria para el tratamiento de datos.

En el desarrollo de aplicaciones en general, muchas veces te encuentras con la duda de si la lógica de negocio lo programamos en la aplicación, por ejemplo, con Java, o en la base de datos con procedimientos almacenados. Tal vez aplicar una solución mixta, parte de la lógica en la aplicación y parte en la base de datos.

La posibilidad de que desarrolles las reglas de negocio en procedimientos almacenados puede representar muchas ventajas, por ejemplo, si hay algún cambio en la regla de negocio, puede que sea suficiente la actualización el procedimiento almacenado, y no harías ningún cambio en la aplicación.

Al estudiar este curso te estás preparando para que puedas desempeñarte como programador de base de datos SQL Server.

Eric Gustavo Coronel Castillo
INSTRUCTOR

Índice

BASES DE DATOS

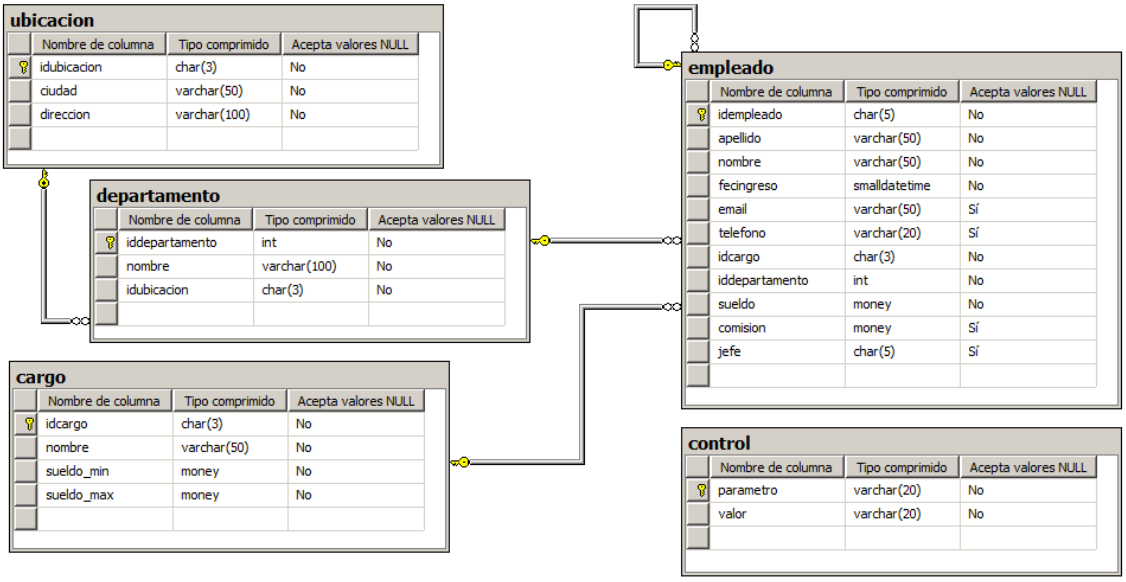
OBTENER SCRIPTS

Para obtener los scripts de las bases de datos utilizadas en la presente separata utiliza la siguiente URL:

https://github.com/gcoronelc/databases

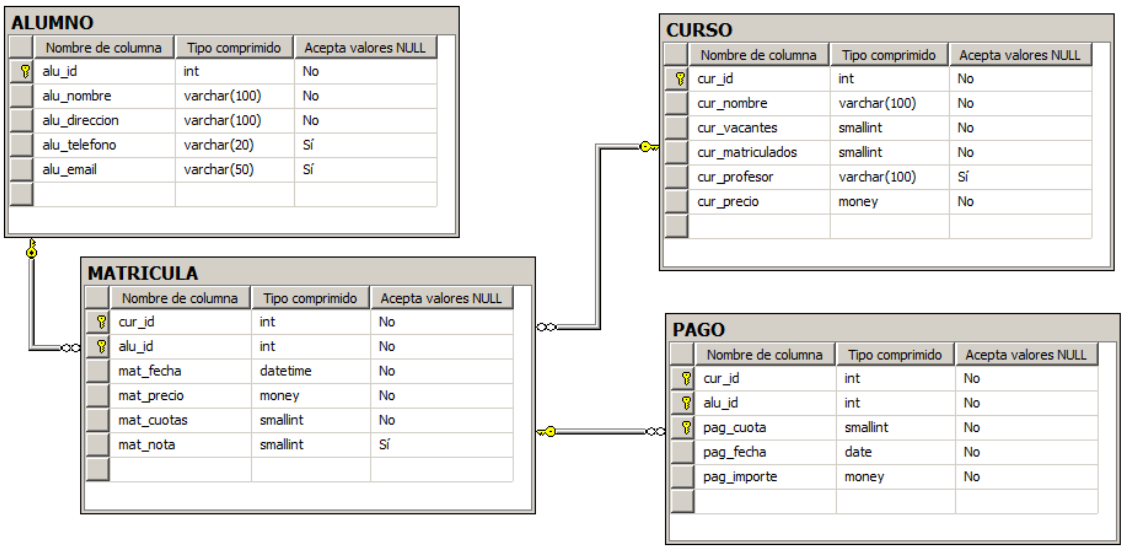
BASE DE DATOS DE RECURSOS HUMANOS – RH

Base de datos básica de recursos humanos.



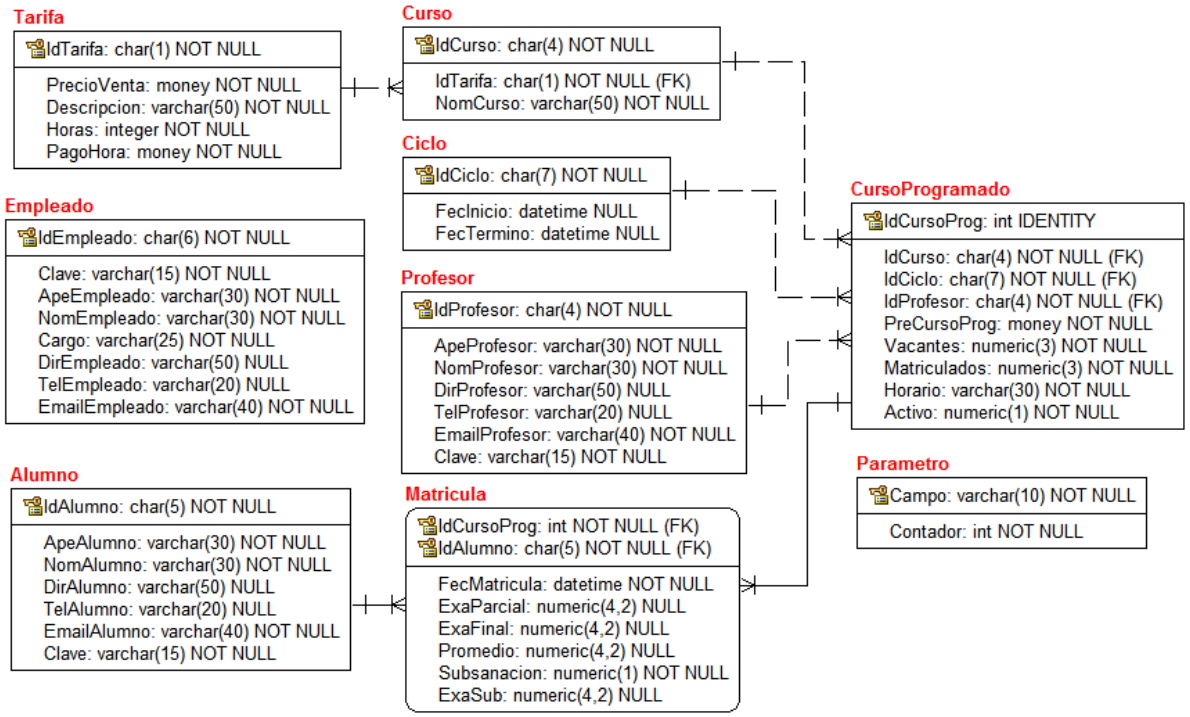
BASE DE DATOS ACADÉMICA – EDUCA

Base de datos bastante simple de gestión académica.



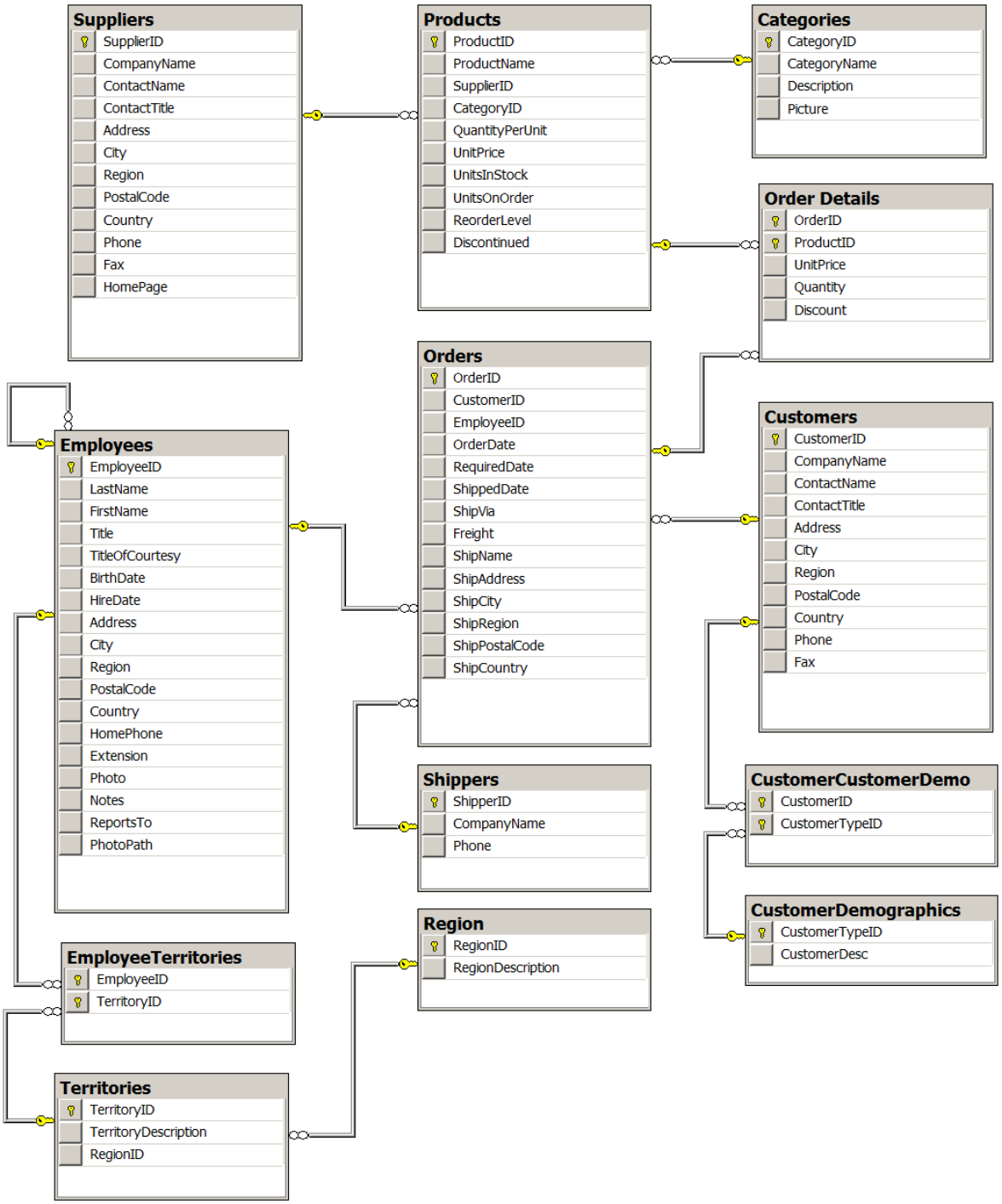
BASE DE DATOS ACADÉMICA – EDUTEC

Base de datos de gestión de cursos cortos.



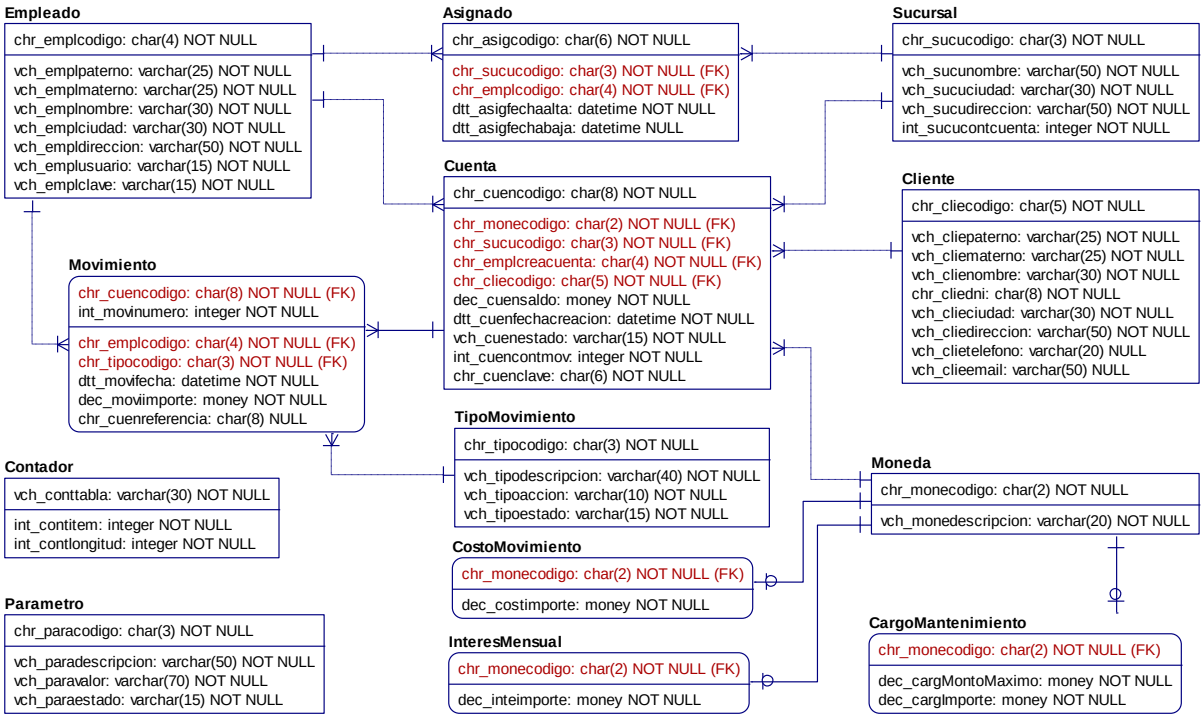
BASE DE DATOS DE COMERCIAL – NORTHWIND

Base de datos comercial clásica de SQL Server.



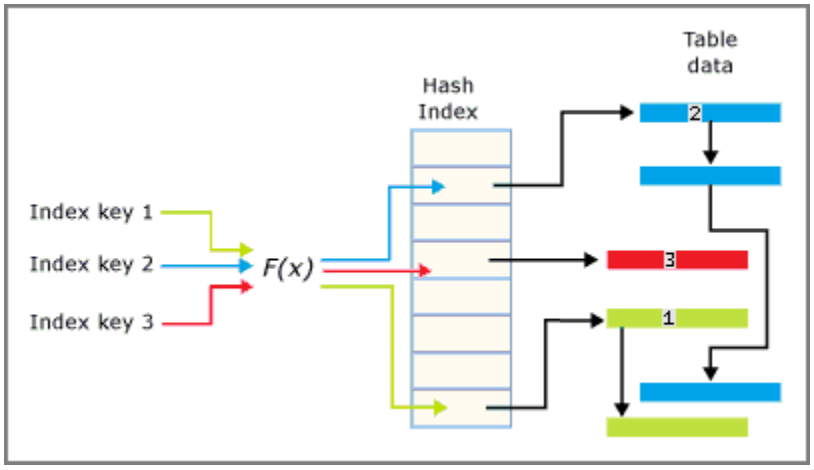
BASE DE DATOS EUREKABANK

Modelo de base de datos de gestión de cuentas de ahorro de una institución financiera.



Capítulo 1
GESTIÓN DE INDICES

CONCEPTO



Un índice es una estructura que proporciona acceso rápido a las filas de una tabla en base a los valores de una o más columnas (clave).

Un índice es un conjunto de valores claves y apuntadores lógicos que permite ejecutar búsqueda de registros de modo similar a la manera como buscamos un tema en el índice analítico de un libro. Por lo general, todas las consultas se ejecutan más rápido cuando se utilizan índices.

ACCESO A LOS DATOS

SQL Server accede a los datos de una de estas dos maneras:

- Examinando todas las páginas de datos de la tabla. Este proceso se conoce como **Table Scan**. Cuando SQL Server realiza un Table Scan, esto es lo que sucede:
  - ✓ Lee desde el principio de la tabla.
  - ✓ Examina página a página todas las filas de la tabla.
  - ✓ Extrae las filas que corresponden al criterio de la consulta.
- Usando los índices. Este proceso se conoce como **Index Seek**. Cuando SQL Server usa un índice, esto es lo que sucede:

✓ Atraviesa la estructura de árbol del índice para encontrar las filas que la consulta requiere.

✓ Extrae solo las filas que se corresponden con el criterio de la consulta.

Cuando se envía una consulta, SQL Server determina primero si un índice existe. Entonces, el **Query Optimizer**, el componente responsable de generar el plan de ejecución óptimo para la consulta, determina si es más eficaz examinar la tabla o utilizar el índice para acceder a los datos.

CRITERIOS PARA CREAR ÍNDICES

Si estás considerando crear un índice, aquí tienes algunas recomendaciones.

Razones para crear índices

Los índices aceleran la recuperación de los datos. Por ejemplo, sin ningún índice, tendrías que recorrer todas las páginas de un libro hasta encontrar la información que estamos buscando.

- Refuerzan la unicidad de las filas.
- Incrementan la velocidad de recuperación de datos:
  - ✓ Los joins se ejecutan más rápido si la columna llave foránea está indexada.
  - ✓ Las consultas ORDER BY y GROUP BY se ejecutan más rápido.

Razones para no crear índices

En general, cuando ejecutamos operaciones de lectura, los índices favorecen el proceso; cuando las operaciones son de escritura, los índices hacen que el rendimiento del sistema disminuya.

- Consumen espacio de disco.
- Producen sobrecarga en el sistema. Cuando se modifican datos de columnas indexadas, el índice es actualizado automáticamente para reflejar los cambios.

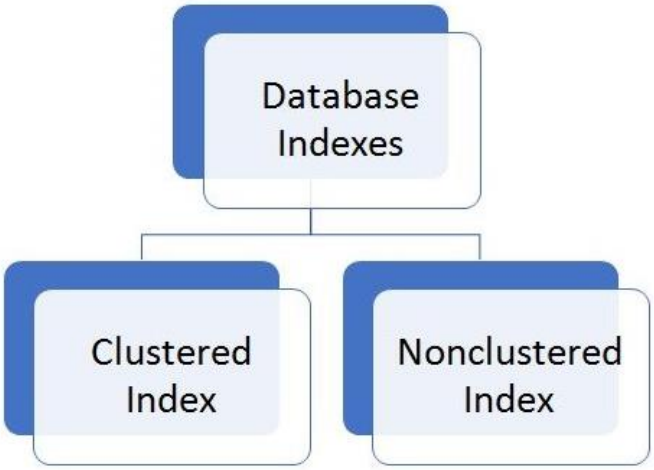
Columnas a indexar

- Clave primaria.
- Clave foránea.
- Columnas en las que se busca por rango de valores.
- Columnas utilizadas como criterio de ordenamiento.

Columnas que no deben indexarse

- Columnas en las que no se ejecuta búsqueda.
- Columnas con pocos valores únicos o que retornan un gran porcentaje de filas.
- Columnas de tipo bit, text o image.

TIPOS DE ÍNDICES



Hay dos opciones para el almacenamiento físico de sus índices:

- índice agrupado (clustered)
- índice no agrupado (nonclustered)

Indice clustered

- Ordena físicamente la tabla. Las filas de la tabla se ordenan según el orden de los valores claves del índice clustered.
- Solo se puede definir un índice clustered por cada tabla.

Indice nonclustered

- Es el tipo de índice por defecto.
- Se reconstruyen automáticamente cuando se crea, se elimina o se redefine el índice clustered.

Observación:

- Se pueden definir hasta 249 índices por tabla.
- Siempre crear el índice clustered antes que los índices nonclustered.

CREACIÓN DE INDICES

Sintaxis

```
CREATE [UNIQUE] [CLUSTERED | NONCLUSTERED]
INDEX NombreDelIndice
ON NombreDeTabla(Columna1, Columna2, ...)
[ FILLFACTOR = <factor de relleno> ]
```

Para mayor información consultar:

```
https://docs.microsoft.com/en-us/sql/t-sql/statements/create-index-transact-sql
```

Creación de índice CLUSTERED

Ejemplo 1

```
CREATE CLUSTERED INDEX EMPLEADO_IDXC
ON EMPLEADO(ID_PERSONA)
```

Creación de índice NONCLUSTERED

Ejemplo 2

```
CREATE INDEX EMPLEADO_IDXN
ON EMPLEADO(NOMBRE)
```

Creación de índice UNIQUE

El índice UNIQUE es aquel que no permite repetición de los valores que conforman el índice.

Ejemplo 3

```
CREATE UNIQUE INDEX EMPLEADO_IDXU
ON EMPLEADO(EMAIL)
```

MANTENIMIENTO DE INDICES

Borrar un índice

Sintaxis

```
DROP INDEX  NombreDeTabla.NombreDeIndice
```

Regenerar un índice

```
ALTER INDEX <index name> ON <table name> REBUILD
```

Regenerar los índices de una tabla

```
ALTER INDEX ALL ON <table name> REBUILD
```

Fragmentación de índices

```
SELECT
   c.name "Table name",
   b.name "Index",
   avg_fragmentation_in_percent "Frag (%)",
   page_count "Page count"
FROM sys.dm_db_index_physical_stats (DB_ID(), NULL, NULL, NULL, NULL ) AS a
JOIN sys.indexes AS b ON a.object_id = b.object_id AND a.index_id = b.index_id
JOIN sys.tables c ON b.object_id = c.object_id
ORDER BY  3 DESC;
GO
```

Reorganizar índices

Sintaxis

```
ALTER INDEX <index name> ON <table name> REORGANIZE
```

Reorganizar todos los índices de una tabla

Sintaxis

```
  ALTER INDEX ALL ON <table name> REORGANIZE
```

Capítulo 2
FUNDAMENTOS GENERALES

INTRODUCCIÓN

Transact-SQL o T-SQL es un lenguaje muy potente que te permite programar casi cualquier tarea que quieras efectuar sobre la base de datos, va más allá de lenguaje SQL, características que permiten definir la lógica necesaria para el tratamiento de datos.

T-SQL permite

Con T-SQL puedes:

- Definir bloques de instrucciones SQL que se tratan como unidades de ejecución.
- Realizar ejecuciones condicionales.
- Realizar ejecuciones iterativas o repetitivas.
- Garantizar el tratamiento modular con la declaración de variables locales y el uso de procedimientos almacenados.
- Manipular tupla a tupla el resultado de una consulta.
- Etc.

T-SQL no permite

Hay algunas cosas que no podrás hacer:

- Crear interfaces de usuario.
- Crear aplicaciones ejecutables, sino elementos que en algún momento llegarán al servidor de datos y serán ejecutados.

Debido a estas restricciones utilizaras T-SQL para crear funciones de usuario, procedimientos almacenados y triggers.

REGLAS DE FORMATO DE LOS IDENTIFICADORES

Los identificadores son los nombres de los objetos: servidores, bases de datos, tablas, vistas, columnas, índices, funciones, procedimientos, triggers, restricciones, reglas, etc.

Las reglas de formato de los identificadores normales dependen del nivel de compatibilidad de la base de datos, utilizando la cláusula SET COMPATIBILITY_LEVEL de la instrucción ALTER TABLE. Cuando el nivel de compatibilidad es 90, (el asignado por defecto) se aplican las reglas siguientes para los nombres de los identificadores:

- No puede ser una palabra reservada.
- El nombre debe tener entre 1 y 128 caracteres, excepto para algunos tipos de objetos en los que el número es más limitado.
- El nombre debe empezar por:
    1. Una letra, como aparece definida por el estándar Unicode 3.2. La definición Unicode de letras incluye los caracteres latinos de la "a" a la "z" y de la "A" a la "Z".
    2. El carácter de subrayado (_), arroba (@) o número (#).
- Ciertos símbolos al principio de un identificador tienen un significado especial en SQL Server. Un identificador que empieza con el signo de arroba indica un parámetro o una variable local. Un identificador que empieza con el signo de número indica una tabla o procedimiento temporal. Un identificador que empieza con un signo de número doble (##) indica un objeto temporal global.
- Algunas funciones de Transact-SQL tienen nombres que empiezan con un doble signo de arroba (@@). Para evitar confusiones con estas funciones, se recomienda no utilizar nombres que empiecen con @@.
- No se permiten los caracteres especiales o los espacios incrustados.

Si quieres utilizar un nombre que no siga estas reglas, normalmente para poder incluir espacios en blanco, lo tienes que escribir encerrado entre corchetes (también se pueden utilizar las comillas, pero se recomienda utilizar los corchetes).

LAS EXPRESIONES

Una expresión es una combinación de símbolos y operadores que el motor de base de datos de SQL Server evalúa para obtener un único valor. Una expresión simple puede ser una sola constante, variable, columna o función escalar.

Los operadores se pueden usar para combinar dos o más expresiones simples y formar una expresión compleja. Dos expresiones pueden combinarse mediante un operador si ambas tienen tipos de datos admitidos por el operador y se cumple al menos una de estas condiciones:

- Las expresiones tienen el mismo tipo de datos.
- El tipo de datos de menor prioridad se puede convertir implícitamente al tipo de datos de mayor prioridad.
- La función CAST puede convertir explícitamente el tipo de datos con menor prioridad al tipo de datos con mayor prioridad o a un tipo de datos intermedio que pueda convertirse implícitamente al tipo de datos con la mayor prioridad.

Tipos de operadores

1. Operadores numéricos
2. Operadores bit a bit: realizan manipulaciones de bits entre dos expresiones de cualquiera de los tipos de datos de la categoría del tipo de datos entero.
3. Operadores de comparación.
4. Operadores lógicos.
5. Operadores de cadenas.

Resultados de la expresión

Si se combinan dos expresiones mediante operadores de comparación o lógicos, el tipo de datos resultante es booleano y el valor es uno de los siguientes: TRUE, FALSE o UNKNOWN.

1. Cuando dos expresiones se combinan mediante operadores aritméticos, bit a bit o de cadena, el operador determina el tipo de datos resultante.
2. Las expresiones complejas formadas por varios símbolos y operadores se evalúan como un resultado formado por un solo valor. El tipo de datos, intercalación, precisión y valor de la expresión resultante se determina al

combinar las expresiones componentes de dos en dos, hasta que se alcanza un resultado final. La prioridad de los operadores de la expresión define la secuencia en que se combinan las expresiones.

OTROS ELEMENTOS DEL LENGUAJE

Comentarios

Como en cualquier otro lenguaje de programación, se utilizan comentarios destinados a facilitar la legibilidad del código.

Sintaxis

```
-- Este es un comentario
```

BEGIN...END

Encierra un conjunto de instrucciones Transact-SQL de forma que estas instrucciones formen un bloque de instrucciones.

Sintaxis

```
BEGIN

   -- Instrucciones

END
```

Capítulo 3
FUNDAMENTOS DE PROGRAMACIÓN

BLOQUE ANÓNIMO

Bloque de instrucciones no nombrado y no graba en la base de datos.

Sintaxis

```
BEGIN

   Sentencias a ejecutar

END;
GO
```

Ejemplo 4

```
BEGIN

   DECLARE  @NUM1 INT,  @NUM2 INT,  @SUMA INT;

   SET @NUM1 =  CAST( RAND() * 100 AS INT );
   SET @NUM2 =  CAST( RAND() * 100 AS INT );

   SET @SUMA = @NUM1 + @NUM2;

   PRINT CONCAT( 'NUM1 = ', @NUM1 );
   PRINT CONCAT( 'NUM2 = ', @NUM2 );
   PRINT CONCAT( 'SUMA = ', @SUMA );

END;
GO
```

FUNCIONES

Función Escalar

Sintaxis

```
CREATE FUNCTION [ esquema. ] nombre_funcion
(
   [  @parametro tipo_dato [ = valor_defecto ] [ READONLY ]
      [ ,...n ]
   ]
)
RETURNS tipo_dato_retorno
[ AS ]
BEGIN
   cuerpo_función
   RETURN expression;
END;
```

Ejemplo 5

```
CREATE FUNCTION dbo.fn_suma ( @num1 int, @num2 int )
RETURNS int
AS
BEGIN
   DECLARE @suma int;
   SET @suma = @num1 + @num2;
   RETURN @suma;
END;
GO

SELECT dbo.fn_suma( 24, 56 ) as suma;
GO

suma
-----------
80
```

Función de tabla en línea

Sintaxis

```
CREATE FUNCTION [ esquema. ] nombre_función
(
   [  @parametro tipo_dato [ = valor_defecto ] [ READONLY ]
      [ ,...n ]
   ]
)
RETURNS TABLE
[ AS ]
RETURN [ ( ] sentencia_select [ ) ] ;
```

Ejemplo 6

```
USE RH;
GO

CREATE FUNCTION dbo.fn_empleados ( @p_dpto int )
RETURNS TABLE
AS
RETURN
   SELECT idempleado, apellido, nombre
   FROM dbo.empleado
   WHERE iddepartamento = @p_dpto;
GO

SELECT * FROM dbo.fn_empleados(103);
GO
```

Función de tabla de múltiples instrucciones

Sintaxis

```
CREATE FUNCTION [ esquema. ] nombre_función
(
   [  @parametro tipo_dato [ = valor_defecto ] [ READONLY ]
      [ ,...n ]
   ]
)
RETURNS @return_variable TABLE definición_tipo_tabla
[ AS ]
BEGIN
   cuerpo_funcion
   RETURN;
END;
```

Ejemplo 7

```
CREATE FUNCTION dbo.fn_catalogo ( )
RETURNS @tabla TABLE
(
   codigo int identity(1,1) primary key not null,
   nombre varchar(50) not null,
   precio money not null
)
AS
BEGIN
   INSERT INTO @tabla(nombre,precio) values('Televisor', 1500.00);
   INSERT INTO @tabla(nombre,precio) values('Refrigeradora', 1450.00);
   INSERT INTO @tabla(nombre,precio) values('Lavadora', 1350.00);
   RETURN;
END;
GO

SELECT * FROM dbo.fn_catalogo();
GO
```

Ejemplo 8

```
USE RH;
GO

CREATE FUNCTION dbo.fn_planilla ( )
RETURNS @planilla TABLE
(
   codigo int primary key not null,
   nombre varchar(50) not null,
   plan_actual money not null,
   plan_proyectada money not null
)
AS
BEGIN
   INSERT INTO @planilla
   SELECT
      d.iddepartamento as codido,
      d.nombre as nombre,
      SUM(e.sueldo) as "planilla actual",
      cast(SUM(e.sueldo * 1.15) as money) as "planilla proyectada"
   FROM dbo.departamento as d
   JOIN dbo.empleado as e
   ON d.iddepartamento = e.iddepartamento
   GROUP BY d.iddepartamento, d.nombre;
   RETURN;
END;
GO

SELECT * FROM dbo.fn_planilla ( )
GO
```

PROCEDIMIENTOS

Sintaxis

```
CREATE { PROC | PROCEDURE } [ esquema.] nombre_procedimiento
   [ @parametro tipo_dato [ = default ] [ OUT | OUTPUT ] [READONLY] ]
   [ ,...n ]
AS
BEGIN

   cuerpo del procedimiento

END;
```

Ejemplo 9: Procedimiento simple

```
USE EDUCA;
GO

CREATE PROCEDURE dbo.usp_lista_cursos
AS
BEGIN
   SET NOCOUNT ON;
   SELECT * FROM dbo.curso;
END;
GO

EXEC dbo.usp_lista_cursos;
GO
```

Ejemplo 10: Procedimiento con parámetros de entrada

```
CREATE PROCEDURE dbo.usp_suma ( @num1 int, @num2 int )
AS
BEGIN
   DECLARE @suma int;
   SET @suma = @num1 + @num2;
   SELECT @num1 NUM1, @num2 NUM2, @suma SUMA;
END;
GO

EXEC dbo.usp_suma 54, 76;
GO
```

Ejemplo 11: Procedimiento con parámetro de salida

```
USE EDUCA;
GO

CREATE PROCEDURE dbo.usp_precio ( @p_idcurso int, @p_precio money OUT )
AS
BEGIN
   SELECT @p_precio = cur_precio
   FROM dbo.CURSO
   WHERE cur_id = @p_idcurso;
END;
GO

BEGIN
   DECLARE @precio money;
   EXEC dbo.usp_precio 3, @precio OUT;
   PRINT CONCAT( 'PRECIO: ', @precio );
END;
GO

PRECIO: 1815.00
```

ELEMENTOS DE PROGRAMACIÓN

Variables

Sintaxis:

```
DECLARE
   @nombre_variable [AS] data_type [ = value ] [ , ... ]
```

Sentencia de asignación

Instrucción SET

```
SET @nombre_variable = expresión;

SET @nombre_variable = ( sentencia_select );
```

Sentencia SELECT

```
SElECT @nombre_variable = ( sentencia SELECT ), . . .
FROM . . .
```

EJERCICIOS PROPUESTOS

A continuación, tienes una lista de ejercicios propuestos que debes desarrollar con un compañero de clase.

1. Desarrolle una función que permita calcular el promedio de 3 números.
2. Desarrolle una función que reporte de la cantidad de alumnos matriculados y las vacantes disponibles de un determinado curso. Base de datos EDUTEC.
3. Desarrolle una función que dado el código de un curso reporte los alumnos que tienes saldo pendiente. Base de datos EDUCA.
4. Desarrolle un procedimiento que reporte por cada curso la cantidad de alumnos matriculados, el importe recaudado y el importe que falta cobrar.
5. Desarrollar un procedimiento que permita obtener la cantidad de artículos vendidos de una determinada categoría. Base de datos: NORTHWIND.
6. Desarrollar una función que permita convertir soles a dólares y viceversa.

Capítulo 4
ESTRUCTURAS DE CONTROL

BLOQUE

Sintaxis

```
BEGIN

   sentencias

END;
```

ESTRUCTURAS CONDICIONALES

Estructura: IF

Sintaxis

```
IF expresión_logica
   { sentencia | bloque }
[ ELSE
   { sentencia | bloque } ]
```

Ejercicio 1

Desarrollar una función que retorne el mayor de 3 números.

Ejercicio 2

Desarrollar una función que permita obtener la condición de APROBADO o DESAPROBADO de un estudiante en función a su nota final.

Estructura: CASE

Caso 1: Simple CASE expression

```
CASE expresión_entrada
     WHEN expresion1 THEN resultado1
     [ ...n ]
     [ ELSE resultado_else ]
END;
```

Caso 2: Searched CASE expression

```
CASE
     WHEN expresión_logica_1 THEN resultado1
     [ ...n ]
     [ ELSE resultado_else ]
END;
```

Ejercicio 3

Desarrollar una función que simule una calculadora para las operaciones básicas: Suma, Resta, Multiplicación y División.

Ejercicio 4

Desarrollar una función que me permita averiguar a qué estación del año corresponde una fecha.

Ejercicio 5

Desarrollar una función que permita obtener la condición de un estudiante en función a su promedio final y su porcentaje de asistencia según el siguiente cuadro.

ASISTENCIA (%)	PROMEDIO FINAL	CONDICIÓN
[80, ∞>	[14, 20]	APROBADO
[80, ∞>	[10, 14>	ASISTENTE
Otros casos	Otros casos	DESAPROBADO

Ejercicio 6

Desarrollar una función que permita encontrar la clasificación del sueldo de un empleado según el siguiente cuadro:

RANGO DEL SUELDO	CLASIFICACION
[0, 2000]	BAJO
<2000, 4500]	REGULAR
<4500, ∞>	BUENO

Ejercicio 7

La empresa ha decidido otorgar una bonificación. Esta bonificación es un porcentaje de su sueldo y depende de la escala de su sueldo:

ESCALA	% BONIFICACIÓN
BUENO	50%
REGULAR	70%
BAJO	100%

Se debe trabajar con la base de datos RECURSOS.

Diseñe he implemente la solución más adecuada.

Ejercicio 8

Desarrollar una función que calcule el promedio de 4 notas, pero que solo considere las 3 mejores.

ESTRUCTURAS DE BUCLE

Estructura WHILE

Sintaxis

```
WHILE expresión_lógica
   sentencia | bloque ;
```

Sentencia BREAK

Sintaxis

```
BREAK;
```

Sentencia CONTINUE

Sintaxis

```
CONTINUE;
```

Sentencia GOTO

Etiqueta

```
ETIQUETA:
```

Salto incondicional

```
GOTO ETIQUETA;
```

Ejercicio 9

Desarrollar una función para calcular el factorial de un número.

Ejercicio 10

Desarrollar un procedimiento para calcular el MCD y MCM de dos números.

Ejercicio 11

Desarrollar un procedimiento que imprima los "N" primeros términos de la serie de Fibonacci.

Capítulo 5
GESTIÓN DE DATOS

INSERTANDO DATOS

Sentencia INSERT

```
[ WITH <common_table_expression> [ ,…n ] ]
INSERT
[ TOP ( expression ) [ PERCENT ] ]
[<OUTPUT Clause>]
INTO nombre_tabla [ ( columnas ) ]
VALUES ( valores ) | Instrucción_SELECT
```

Insertar una sola fila de datos

```
USE RH;
GO

INSERT INTO DBO.ubicacion
VALUES('U05','CHICLAYO','Av. Balta 1543 – Cercado');
GO

SELECT * FROM DBO.ubicacion
GO
```

No es necesario especificar los nombres de columna en la lista de columnas porque se está suministrando valores para todas las columnas y en el mismo orden que se encuentran en la tabla.

Insertar varias filas de datos

```
USE EDUCA;
GO

INSERT INTO ALUMNO(alu_id, alu_nombre, alu_direccion, alu_telefono, alu_email)
VALUES
(100, 'LLERENA BOLIVAR, PAMELA','LA MOLINA', '982354768', 'ollerena@gmail.com'),
(101, 'SALAZAR MENDO, AMALIA IRENE', 'MIRAFLORES', NULL, 'asalazar@peru.com'),
(102, 'VELASQUEZ RAMOS, MARIA EULALIA','SAN BORJA',NULL,'mvelasquez@gmail.com');
GO

SELECT * FROM DBO.ALUMNO;
GO
```

Insertar datos en una tabla con una columna identidad

```
USE EDUCA;
GO

SET IDENTITY_INSERT DBO.ALUMNO ON;
GO

INSERT INTO ALUMNO(alu_id, alu_nombre, alu_direccion, alu_telefono, alu_email)
VALUES(200,'AYALA FERNANDEZ, VALERIA','SURCO','875698456','vayala@hotmail.com');
GO

SET IDENTITY_INSERT DBO.ALUMNO OFF;
GO

SELECT * FROM DBO.ALUMNO
ORDER BY 1 DESC;
GO
```

Usar TOP para limitar los datos insertados de la tabla origen

```
USE EDUCA;
GO

IF OBJECT_ID ('DBO.ALUMNOS2', 'U') IS NOT NULL
    DROP TABLE DBO.ALUMNOS2;
GO

CREATE TABLE dbo.alumnos2
( codigo   int NOT NULL,
  nombre   varchar(100) NOT NULL,
  email    varchar(50) NOT NULL
);
GO

INSERT TOP (4) INTO DBO.alumnos2(codigo,nombre,email)
SELECT alu_id, alu_nombre, alu_email FROM DBO.ALUMNO;
GO

SELECT * FROM DBO.alumnos2;
GO
```

Para ver las filas que se están insertando debemos utilizar la cláusula OUTPUT:

```
INSERT TOP (4) INTO DBO.alumnos2(codigo,nombre,email)
OUTPUT inserted.codigo, inserted.nombre, inserted.email
SELECT alu_id, alu_nombre, alu_email FROM DBO.ALUMNO;
GO
```

Ejercicio 1

En la base de datos **RH** crear una tabla de nombre PLANILLA que permita guardar el importe de la planilla por puesto de trabajo en cada departamento, la información a registrar por fila es la siguiente:

- Código de departamento
- Nombre de departamento
- Código de puesto de trabajo
- Nombre del puesto de trabajo
- Cantidad de trabajadores
- Importe de la planilla sin comisión
- Importe de la planilla con comisión

Luego, construya una sentencia INSERT para llenar la tabla **PLANILLA**.

Ejercicio 2

En la base de datos EDUCA crear la tabla RESUMEN que permita registrar la siguiente información por curso:

- Código del curso
- Nombre del curso
- Cantidad de matriculados
- Importe comprometido según las matriculas
- Importe recaudado según los pagos
- Cantidad de alumnos aprobados (Nota >= 13)
- Cantidad de desaprobados (Nota < 13)
- Cantidad de ausentes (Nota = NULL>

Luego construya una sentencia **INSERT** para llenar la tabla **RESUMEN**.

ACTUALIZANDO DATOS

Sentencia UPDATE

```
[ WITH <common_table_expression> […n] ]
UPDATE
[ TOP ( expression ) [ PERCENT ] ]
nombre_tabla
SET column_name = expression [,  . . . ]
[ <OUTPUT Clause> ]
[ FROM{ <table_source> } [ ,…n ] ]
[ WHERE <search_condition> ]
```

Usar una instrucción UPDATE simple

En el ejemplo siguiente se actualiza un solo valor de columna para todas las filas de la tabla **dbo.CURSO**.

```
USE EDUCA;
GO

UPDATE dbo.CURSO
SET cur_precio = ROUND(cur_precio * 1.10,0)
```

Actualizar varias columnas

En el siguiente ejemplo se actualizan los valores de las columnas cur_vacantes, y cur_precio para todas las filas de la tabla CURSO.

```
USE EDUCA;
GO

UPDATE dbo.CURSO
SET  cur_vacantes = cur_vacantes + 2,
     cur_precio = ROUND(cur_precio * 1.10,0)
```

Usar la cláusula WHERE

En el ejemplo siguiente se utiliza la cláusula WHERE para especificar la fila que se va a actualizar. La instrucción actualiza el valor de la columna cur_precio de la tabla **CURSO** para todas las filas que corresponde al curso **SQL Server Administración**, la condición es **cur_id=2**.

```
USE EDUCA;
go

SELECT cur_id, cur_precio
FROM dbo.CURSO where cur_id = 2;
go

cur_id      cur_precio
----------- ---------------------
2           1210,00

UPDATE dbo.CURSO
SET cur_precio = 1500.00
WHERE cur_id = 2;

SELECT cur_id, cur_precio
FROM dbo.CURSO where cur_id = 2;
go

cur_id      cur_precio
----------- ---------------------
2           1500,00
```

Usar la cláusula TOP

En los siguientes ejemplos use la cláusula TOP para limitar el número de filas que se modifican en una instrucción UPDATE.

Cuando se usa una cláusula TOP (n) con UPDATE, la operación de actualización se realiza en una selección aleatoria de un número de filas 'n'.

En el siguiente ejemplo se incrementa en un 25 por ciento el sueldo de 5 empleados seleccionados en forma aleatoria. En el resultado, la columna old muestra el sueldo antes de la actualización y la columna new el sueldo después de la actualización.

```
USE RH;
go

UPDATE TOP (5) dbo.empleado
SET sueldo = sueldo * 1.10
OUTPUT deleted.idempleado, deleted.sueldo old, inserted.sueldo new;

idempleado old                  new
---------- -------------------- --------------------
E0001      25000,00             27500,00
E0002      8000,00              8800,00
E0003      15000,00             16500,00
E0004      1800,00              1980,00
E0005      7000,00              7700,00

(5 filas afectadas)
```

Si necesita usar TOP para aplicar actualizaciones por orden cronológico, o por algún otro criterio de ordenamiento, debe utilizarla junto con ORDER BY en una subconsulta.

En el siguiente ejemplo se incrementa en un 25 por ciento el sueldo de 5 empleados que tienen el menor salario en la empresa. En el resultado, la columna **old** muestra el sueldo antes de la actualización y la columna **new** el sueldo después de la actualización.

```
USE RH;
go

UPDATE dbo.empleado
SET sueldo = sueldo * 1.10
OUTPUT deleted.idempleado, deleted.sueldo old, inserted.sueldo new
FROM (SELECT TOP 5 idempleado FROM dbo.empleado
      order by sueldo asc) AS t
WHERE dbo.empleado.idempleado = t.idempleado;

idempleado old                  new
---------- -------------------- ----------------------
E0004      1980,00              2178,00
E0011      2000,00              2200,00
E0018      2000,00              2200,00
E0015      2500,00              2750,00
E0014      3000,00              3300,00

(5 filas afectadas)
```

Usar la cláusula WITH common_table_expression

En el siguiente ejemplo se está agregando la columna **cur_recaudado** a la tabla **CURSO** para almacenar el importe recaudado por los pagos efectuados de los alumnos, utilizando la cláusula **WITH** se está construyendo una sentencia **SELECT** para obtener el importe recaudado por curso que luego es utilizada en la sentencia **UPDATE**. La cláusula OUTPUT muestra los cambios realizados.

```
USE EDUCA;
GO

ALTER TABLE dbo.CURSO
ADD cur_recaudado money NOT NULL DEFAULT 0.0;
GO

WITH recaudado(cur_id, importe) as
(
   SELECT cur_id, sum(pag_importe)
   FROM dbo.PAGO
   GROUP BY cur_id
)
UPDATE dbo.CURSO
SET cur_recaudado = recaudado.importe
OUTPUT  deleted.cur_id, deleted.cur_recaudado old, inserted.cur_recaudado new
FROM recaudado
WHERE dbo.CURSO.cur_id = recaudado.cur_id;
GO

cur_id      old                   new
----------- --------------------- ---------------------
1           0,00                  1800,00
2           0,00                  3310,00

(2 filas afectadas)
```

Especificar una subconsulta en la cláusula SET

En el siguiente ejemplo se usa una subconsulta en la cláusula **SET** para determinar el valor que se utilizará para actualizar la columna. La subconsulta debe devolver solo un valor escalar. Es decir, un solo valor.

En el siguiente ejemplo se está utilizando la base de datos EDUCA para crear la tabla INGRESOS, el propósito de esta tabla es almacenar el importe de la suma de los ingresos por curso. Se está utilizando una subconsulta para actualizar la columna **importe**.

```
use EDUCA;
go

IF OBJECT_ID('INGRESOS','U') IS NOT NULL
   DROP TABLE INGRESOS;
GO

SELECT cur_id, cur_nombre, cast(0.0 as money) as importe
INTO dbo.INGRESOS
FROM dbo.CURSO;
go

SELECT * FROM dbo.INGRESOS;
go

cur_id      cur_nombre                                importe
----------- ----------------------------------------- ---------------------
3           Inteligencia de Negocios                  0.00
6           Java Cliente-Servidor                     0.00
5           Java Fundamentos                          0.00
4           Programación Transact-SQL                 0.00
2           SQL Server Administración                 0.00
1           SQL Server Implementación                 0.00

(6 filas afectadas)

update dbo.INGRESOS
set importe = (select SUM(pag_importe)
   from dbo.PAGO
   where dbo.INGRESOS.cur_id = dbo.PAGO.cur_id);
go
```

```
SELECT * FROM dbo.INGRESOS;
go

cur_id      cur_nombre                                 importe
----------- ------------------------------------------ ---------------------
3           Inteligencia de Negocios                   NULL
6           Java Cliente-Servidor                      NULL
5           Java Fundamentos                           NULL
4           Programación Transact-SQL                  NULL
2           SQL Server Administración                  3310.00
1           SQL Server Implementación                  1800.00

(6 filas afectadas)
```

Reto

Cuál sería la modificación a la sentencia UPDATE para que no grabe valores nulos en la columna **importe**.

Ejercicio 3

En la base de datos RH, a la tabla cargo agregarle una columna de nombre **EMPS**, luego utilizando una sentencia UPDATE con subconsulta en esta columna **EMPS** debe guardar la cantidad de empleados por cargo.

ELIMINANDO FILAS

Sentencia DELETE

```
[ WITH <common_table_expression> [ ,...n ] ]
DELETE
[ TOP ( expression ) [ PERCENT ] ]
[ FROM ] nombre_tabla
[ <OUTPUT Clause> ]
[ FROM{ <table_source> } [ ,...n ] ]
[ WHERE <search_condition> ]
```

DELETE sin la cláusula WHERE

En el ejemplo siguiente se eliminan todas las filas de la tabla **alumno2** porque no se utiliza una cláusula WHERE para limitar el número de filas eliminadas.

```
USE EDUCA;
GO

DELETE FROM dbo.alumnos2;
GO
```

Usar la cláusula WHERE para eliminar un conjunto de filas

En el siguiente ejemplo se desarrolla en la base de datos EDUCA, y lo primero que se está realizando es crear una tabla de cursos llamada CURSOS2 con todas las filas de la tabla CURSO para hacer la demostración.

De la tabla CURSOS2 se está eliminando todos los cursos que no tienen ningún alumno matriculado. Finalmente se está eliminando la tabla CURSOS2.

```
USE EDUCA;
go

IF OBJECT_ID('CURSO2','U') IS NOT NULL
   DROP TABLE CURSO2;
GO

SELECT * INTO dbo.CURSO2 FROM dbo.CURSO;
go
```

```
SELECT cur_id, cur_nombre, cur_matriculados FROM dbo.CURSO2;
go

cur_id      cur_nombre                      cur_matriculados
----------- ------------------------------- ----------------
1           SQL Server Implementación       3
2           SQL Server Administración       5
3           Inteligencia de Negocios        0
4           Programación Transact-SQL       0
5           Java Fundamentos                0
6           Java Cliente-Servidor           0

(6 filas afectadas)

DELETE FROM dbo.CURSO2 WHERE cur_matriculados = 0;
go

(4 filas afectadas)

SELECT cur_id, cur_nombre, cur_matriculados FROM dbo.CURSO2;
go

cur_id      cur_nombre                      cur_matriculados
----------- ------------------------------- ----------------
1           SQL Server Implementación       3
2           SQL Server Administración       5

DROP TABLE dbo.CURSO2;
go
```

Usar la cláusula WHERE con una condición compleja

El siguiente ejemplo es muy similar al anterior, la diferencia está en que se eliminan los cursos que no tienen ningún alumno matriculado y que además no tienen profesor asignado, además se está mostrando las filas eliminadas.

```
USE EDUCA;
go

IF OBJECT_ID('CURSO2','U') IS NOT NULL
  DROP TABLE CURSO2;
GO


select * into dbo.CURSO2 from dbo.CURSO;
go
```

```
select cur_id, cur_nombre, cur_profesor, cur_matriculados from dbo.curso2;
go

cur_id  cur_nombre                 cur_profesor             cur_matriculados
------- -------------------------- ------------------------ ------------------
1       SQL Server Implementación  Gustavo coronel          3
2       SQL Server Administración  Gustavo coronel          5
3       Inteligencia de Negocios   Sergio Matsukawa         0
4       Programación Transact-SQL  NULL                     0
5       Java Fundamentos           Gustavo Coronel          0
6       Java Cliente-Servidor      Gustavo Coronel          0

(6 filas afectadas)

delete from dbo.CURSO2
output deleted.cur_id, deleted.cur_profesor, deleted.cur_matriculados
where cur_matriculados = 0 AND cur_profesor is null;
go

cur_id      cur_profesor                  cur_matriculados
----------- ----------------------------- -----------------
4           NULL                          0

(1 filas afectadas)

select cur_id, cur_nombre, cur_profesor, cur_matriculados from dbo.curso2;
go

cur_id  cur_nombre                 cur_profesor             cur_matriculados
------- -------------------------- ------------------------ ------------------
1       SQL Server Implementación  Gustavo coronel          3
2       SQL Server Administración  Gustavo coronel          5
3       Inteligencia de Negocios   Sergio Matsukawa         0
5       Java Fundamentos           Gustavo Coronel          0

6       Java Cliente-Servidor      Gustavo Coronel          0

(5 filas afectadas)

DROP TABLE dbo.CURSO2;
go
```

Utilizar la cláusula TOP para limitar el número de filas eliminadas

Caso 1

En el siguiente ejemplo se está eliminado 5 empleados de manera aleatoria de la tabla EMP2 que se está creando para propósitos de la demostración en la base de datos RH.

```
use rh;
go

IF OBJECT_ID('EMP2','U') IS NOT NULL
   DROP TABLE EMP2;
GO

select * into dbo.emp2 from dbo.empleado;
go

select COUNT(*) emps from dbo.emp2;
go

emps
-----------
22

(1 filas afectadas)

delete top (5) from dbo.emp2
output deleted.idempleado, deleted.nombre;
go

idempleado nombre
---------- --------------------------------------------------
E0001      Gustavo
E0002      Claudia
E0003      Sergio
E0004      Mariela
E0005      Roberto

(5 filas afectadas)
```

```
select COUNT(*) emps from dbo.emp2;
go

emps
-----------
17

(1 filas afectadas)

drop table dbo.emp2;
go
```

Caso 2

Este caso es similar al Caso 1, la diferencia está en que se eliminan los 5 empleados que tienen los mejores sueldos.

```
use rh;
go

IF OBJECT_ID('EMP2','U') IS NOT NULL
   DROP TABLE EMP2;
GO

select * into dbo.emp2 from dbo.empleado;
go

select COUNT(*) emps from dbo.emp2;
go

emps
-----------
22

(1 filas afectadas)

delete from dbo.emp2
output deleted.idempleado, deleted.nombre, deleted.sueldo
where idempleado in ( select top 5 idempleado
        from dbo.emp2 order by sueldo desc );
go
```

```
idempleado nombre                                          sueldo
---------- ----------------------------------------------- ---------------------
E0001      Gustavo                                         25000.00
E0012      Hugo                                            15000.00
E0003      Sergio                                          15000.00
E0009      Ricardo                                         15000.00
E0016      Nora                                            15000.00

(5 filas afectadas)

select COUNT(*) emps from dbo.emp2;
go

emps
-----------
17

(1 filas afectadas)

drop table dbo.emp2;
go
```

Ejercicio 4

En la base de datos RH crear una tabla de empleados auxiliar **EMP2** con todo el contenido de la tabla **EMPLEADO** para desarrollar este ejercicio.

Luego proceda a eliminar de la tabla **EMP2** todos los empleados cuyo sueldo se encuentra fuera del rango según el cargo que desempeña.

COMBINANDO DATOS

Sentencia MERGE

```
[ WITH <common_table_expression> [,...n] ]
MERGE
    [ TOP ( expression ) [ PERCENT ] ]
    [ INTO ] <target_table> [ [ AS ] table_alias ]
    USING <table_source>
    ON <merge_search_condition>
    [ WHEN MATCHED [ AND <clause_search_condition> ]
        THEN <merge_matched> ] [ ...n ]
    [ WHEN NOT MATCHED [ BY TARGET ] [ AND <clause_search_condition> ]
        THEN <merge_not_matched> ]
    [ WHEN NOT MATCHED BY SOURCE [ AND <clause_search_condition> ]
        THEN <merge_matched> ] [ ...n ]
    [ <output_clause> ] ;
```

Usar MERGE para realizar operaciones INSERT y UPDATE

Ejecute el archivo NUEVOS_ALUMNOS.SQL que le será proporcionado por el profesor, este archivo crea una tabla NUEVOS_ALUMNOS en la base de datos EDUCA.

La tabla NUEVOS_ALUMNOS tiene datos de alumnos que deben ser insertados en la tabla ALUMNO, pero algunos de ellos ya están registrados, de los que ya están registrados se debe actualizar las columnas **alu_direccion** y **alu_telefono**, la columna que se debe verificar para saber si se debe hacer un **INSERT** o **UPDATE** es **alu_email**.

```
USE EDUCA;
GO

SELECT * FROM dbo.ALUMNO;
go
```

	alu_id	alu_nombre	alu_direccion	alu_telefono	alu_email
1	1	YESENIA VIRHUEZ	LOS OLIVOS	986412345	yesenia@hotmail.com
2	2	OSCAR ALVARADO FERNANDEZ	MIRAFLORES	NULL	oscar@gmail.com
3	3	GLADYS REYES CORTIJO	SAN BORJA	875643562	gladys@hotmail.com
4	4	SARA RIEGA FRIAS	SAN ISIDRO	NULL	sara@yahoo.com
5	5	JHON VELASQUEZ DEL CASTILLO	LOS OLIVOS	78645345	jhon@mivistar.com
6	6	LLERENA BOLIVAR, PAMELA DEL ROSARIO	LA MOLINA	982354768	ollerena@gmail.com
7	7	SALAZAR MENDO, AMALIA IRENE	MIRAFLORES	NULL	asalazar@peru.com
8	8	VELASQUEZ TORVISCO, MARIA EULALIA	SAN BORJA	NULL	mvelasquez@gmail.com

```
SELECT * FROM dbo.NUEVOS_ALUMNOS;
GO
```

	alu_nombre	alu_direccion	alu_telefono	alu_email
1	YESENIA VIRHUEZ	LA MOLINA	897678567	yesenia@hotmail.com
2	GLADYS REYES CORTIJO	SAN MIGUEL	456879023	gladys@hotmail.com
3	GABRIEL FLORES ARROYO	SAN MIGUEL	435679456	gabriel@gmail.com
4	LUIS ROJAS CASTRO	LOS OLIVOS	546784768	lrojas@hotmail.com
5	WILLY SANCHEZ CACHAY	SAN ISIDRO	345879567	wsanchez@gmail.com
6	SANDRA SOLER GARCIA	SURCO	967435672	ssoler@gmail.com

```
MERGE INTO dbo.alumno AS target
USING (
   SELECT alu_nombre, alu_direccion, alu_telefono, alu_email
   FROM dbo.nuevos_alumnos
) AS source(nombre,direccion,telefono,email)
ON (target.alu_email = source.email)
WHEN MATCHED THEN
   UPDATE SET
      alu_direccion = source.direccion,
      alu_telefono = source.telefono
WHEN NOT MATCHED THEN
     INSERT (alu_nombre,alu_direccion,alu_telefono,alu_email)
     VALUES (source.nombre, source.direccion, source.telefono, source.email )
OUTPUT deleted.*, $action, inserted.*;
```

En el resultado que usted verá en podrá comprobar que algunas filas se actualizaron y otras filas se insertaron como nuevas.

```
SELECT * FROM dbo.ALUMNO;
go
```

	alu_id	alu_nombre	alu_direccion	alu_telefono	alu_email
1	1	YESENIA VIRHUEZ	LA MOLINA	897678567	yesenia@hotmail.com
2	2	OSCAR ALVARADO FERNANDEZ	MIRAFLORES	NULL	oscar@gmail.com
3	3	GLADYS REYES CORTIJO	SAN MIGUEL	456879023	gladys@hotmail.com
4	4	SARA RIEGA FRIAS	SAN ISIDRO	NULL	sara@yahoo.com
5	5	JHON VELASQUEZ DEL CASTILLO	LOS OLIVOS	78645345	jhon@mivistar.com
6	6	LLERENA BOLIVAR, PAMELA DEL ROSARIO	LA MOLINA	982354768	ollerena@gmail.com
7	7	SALAZAR MENDO, AMALIA IRENE	MIRAFLORES	NULL	asalazar@peru.com
8	8	VELASQUEZ TORVISCO, MARIA EULALIA	SAN BORJA	NULL	mvelasquez@gmail.com
9	101	GABRIEL FLORES ARROYO	SAN MIGUEL	435679456	gabriel@gmail.com
10	102	LUIS ROJAS CASTRO	LOS OLIVOS	546784768	lrojas@hotmail.com
11	103	SANDRA SOLER GARCIA	SURCO	967435672	ssoler@gmail.com
12	104	WILLY SANCHEZ CACHAY	SAN ISIDRO	345879567	wsanchez@gmail.com

En el listado anterior también se puede verificar las nuevas filas, así como las actualizaciones realizadas.

Usar MERGE para realizar operaciones UPDATE y DELETE

En el siguiente ejemplo utilizaremos la base de datos **RH**, la demostración se realizará sobre una tabla auxiliar llamada **EMP2**, que inicialmente contiene los mismos datos que la tabla **EMPLEADO**.

Se trata de encontrar el sueldo promedio por departamento, luego debemos eliminar de la tabla EMP2 los empleados que tienen su sueldo menor que el sueldo promedio en su departamento, y los que quedan su sueldo debe incrementarse en 30%.

```
USE RH;
GO

IF OBJECT_ID('EMP2','U') IS NOT NULL
   DROP TABLE EMP2;
GO

SELECT * INTO DBO.emp2 FROM DBO.empleado;
GO

MERGE INTO dbo.EMP2 AS target
USING (
   SELECT iddepartamento , avg(sueldo)
   FROM dbo.empleado
   GROUP BY iddepartamento
) AS source(dpto,sueldo_prom)
ON (target.iddepartamento = source.dpto)
WHEN MATCHED and target.sueldo < source.sueldo_prom THEN
   DELETE
WHEN  MATCHED THEN
```

```
   UPDATE SET
      target.sueldo = target.sueldo * 1.30
OUTPUT deleted.idempleado, deleted.iddepartamento, deleted.sueldo,
   $action, inserted.idempleado, inserted.iddepartamento, inserted.sueldo;

DROP TABLE DBO.EMP2;
```

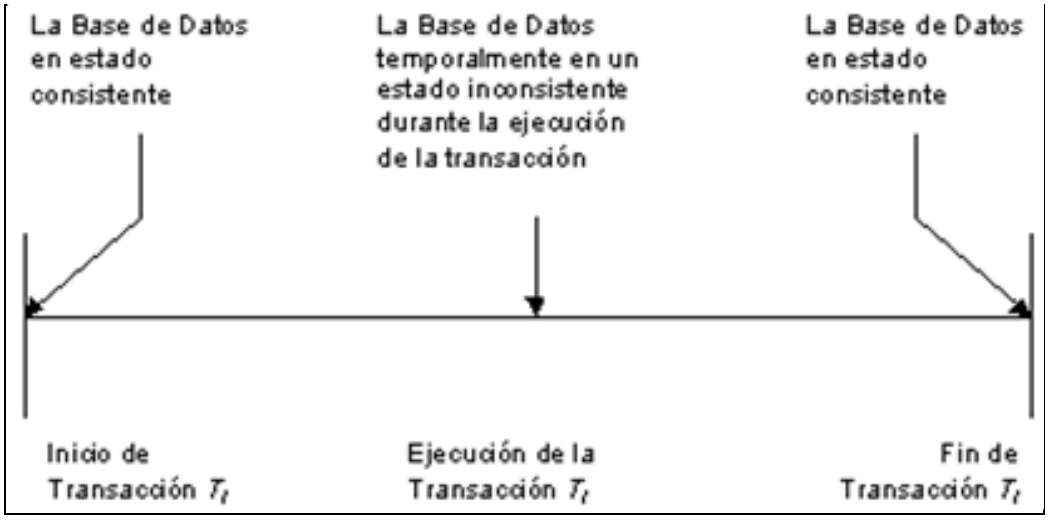
A continuación, parte del resultado de la sentencia MARGE, se puede apreciar que para algunas filas se ejecuta la sentcencia DELETE y para otras la sentencia UPDATE.

	idempleado	iddepartamento	sueldo	$action	idempleado	iddepartamento	sueldo
5	E0011	101	2000,00	DELETE	NULL	NULL	NULL
6	E0003	102	16500,00	UPDATE	E0003	102	21450,00
7	E0004	102	1980,00	DELETE	NULL	NULL	NULL
8	E0005	102	7700,00	UPDATE	E0005	102	10010,00
9	E0006	102	7500,00	UPDATE	E0006	102	9750,00
10	E0007	102	7000,00	DELETE	NULL	NULL	NULL
11	E0008	102	3500,00	DELETE	NULL	NULL	NULL
12	E0016	103	15000,00	UPDATE	E0016	103	19500,00
13	E0017	103	7500,00	UPDATE	E0017	103	9750,00
14	E0018	103	2000,00	DELETE	NULL	NULL	NULL
15	E0019	103	3500,00	DELETE	NULL	NULL	NULL
16	E0020	103	3000,00	DELETE	NULL	NULL	NULL

GESTIÓN DE TRANSACCIONES

Definición

Una transacción es un grupo de acciones que hacen transformaciones consistentes en las tablas preservando la consistencia de la base de datos. Una base de datos está en un estado consistente si obedece todas las restricciones de integridad definidas sobre ella. Los cambios de estado ocurren debido a actualizaciones, inserciones, y eliminaciones de información. Por supuesto, se quiere asegurar que la base de datos nunca entre en un estado de inconsistencia. Sin embargo, durante la ejecución de una transacción, la base de datos puede estar temporalmente en un estado inconsistente. El punto importante aquí es asegurar que la base de datos regresa a un estado consistente al fin de la ejecución de una transacción.



Lo que se persigue con el manejo de transacciones es por un lado tener una transparencia adecuada de las acciones concurrentes a una base de datos y por otro lado tener una transparencia adecuada en el manejo de las fallas que se pueden presentar en una base de datos.

Propiedades de una Transacción

Una transacción debe tener las propiedades ACID, que son las iniciales en inglés de las siguientes características: Atomicity, Consistency, Isolation, Durability.

Atomicidad

Una transacción constituye una unidad atómica de ejecución y se ejecuta exactamente una vez; o se realiza todo el trabajo o nada de él en absoluto.

Coherencia

Una transacción mantiene la coherencia de los datos, transformando un estado coherente de datos en otro estado coherente de datos. Los datos enlazados por una transacción deben conservarse semánticamente.

Aislamiento

Una transacción es una unidad de aislamiento y cada una se produce aislada e independientemente de las transacciones concurrentes. Una transacción nunca debe ver las fases intermedias de otra transacción.

Durabilidad

Una transacción es una unidad de recuperación. Si una transacción tiene éxito, sus actualizaciones persisten, aun cuando falle el equipo o se apague. Si una transacción no tiene éxito, el sistema permanece en el estado anterior antes de la transacción.

Tipos de Transacciones

Transacciones de confirmación automática

El modo de confirmación automática es el modo de administración de transacciones predeterminado de SQL Server. Cada instrucción SQL se confirma o se deshace cuando finaliza. Una conexión de SQL Server funciona en modo de confirmación automática siempre que este modo predeterminado no haya sido sustituido por transacciones explícitas.

Una conexión de SQL Server funciona en modo de confirmación automática hasta que una instrucción **BEGIN TRANSACTION** inicia una transacción explícita. Cuando la transacción explícita se confirma o revierte, SQL Server vuelve al modo de confirmación automática.

Transacciones explicitas

Una transacción explícita es aquella en la que se definen explícitamente el inicio y el final de la transacción.

Las instrucciones SQL que se utilizan son las siguientes:

- **BEGIN TRANSACTION**

  Marca el punto de inicio de una transacción explícita para una conexión.

- **COMMIT TRANSACTION**

  Finaliza correctamente una transacción si no se han encontrado errores. Todos los datos modificados por la transacción se convierten en parte permanente de la base de datos. Se liberan los recursos ocupados por la transacción.

- **ROLLBACK TRANSACTION**

  Borra una transacción en la que se han encontrado errores. Todos los datos modificados por la transacción vuelven al estado en el que estaban al inicio de la transacción. Se liberan los recursos ocupados por la transacción.

Transacciones implícitas

Cuando una conexión funciona en modo de transacciones implícitas, SQL Server Database Engine (Motor de base de datos de SQL Server) inicia automáticamente una nueva transacción después de confirmar o revertir la transacción actual. No tiene que realizar ninguna acción para establecer el inicio de una transacción, sólo tiene que confirmar o revertir cada transacción. El modo de transacciones implícitas genera una cadena continua de transacciones.

Tras establecer el modo de transacciones implícitas en una conexión, la instancia de Motor de base de datos inicia automáticamente una transacción la primera vez que ejecuta una de las siguientes sentencias:

ALTER TABLE	INSERT	CREATE
OPEN	DELETE	REVOKE
DROP	SELECT	FETCH
TRUNCATE TABLE	GRANT	UPDATE

La transacción sigue activa hasta que se ejecute una instrucción **COMMIT** o **ROLLBACK**. Una vez que la primera transacción se ha confirmado o revertido, la

instancia del Motor de base de datos inicia automáticamente una nueva transacción la siguiente vez que la conexión ejecuta una de estas instrucciones. La instancia continúa generando una cadena de transacciones implícitas hasta que se desactiva el modo de transacciones implícitas.

La sentencia SET IMPLICIT_TRANSACTIONS activa o desactiva el modo de transacciones implícitas, su sintaxis es:

```
SET IMPLICIT_TRANSACTIONS { ON | OFF }
```

Capítulo 6
CONTROL DE ERRORES

CONTROL DE ERRORES

Variable: @@ROWCOUNT

Devuelve el número de filas afectadas por la última instrucción. Si el número de filas es mayor de 2 mil millones, use ROWCOUNT_BIG.

```
@@ROWCOUNT
```

Más información en: http://technet.microsoft.com/es-pe/library/ms187316.aspx

Función: ROWCOUNT_BIG ( )

Devuelve el número de filas afectadas por la última instrucción ejecutada. Esta función actúa como @@ROWCOUNT, pero el tipo de valor devuelto de ROWCOUNT_BIG es bigint.

```
ROWCOUNT_BIG ( )
```

Más información en: http://technet.microsoft.com/es-es/library/ms181406.aspx

Variable: @@ERROR

Devuelve el número de error de la última instrucción Transact-SQL ejecutada.

```
@@ERROR
```

Más información en: http://technet.microsoft.com/es-es/library/ms188790.aspx

Función: RAISERROR ( )

Genera un mensaje de error e inicia el procesamiento de errores de la sesión. RAISERROR puede hacer referencia a un mensaje definido por el usuario almacenado en la vista de catálogo **sys.messages** o puede generar un mensaje dinámicamente. El mensaje se devuelve como un mensaje de error de servidor a la aplicación que realiza la llamada o a un bloque CATCH asociado a una estructura TRY…CATCH.

Las nuevas aplicaciones deben utilizar THROW en su lugar.

```
RAISERROR ( { msg_id | msg_str | @local_variable }
   { ,severity ,state }
   [ ,argument [ ,...n ] ] );
```

Más información en: http://msdn.microsoft.com/es-es/library/ms178592.aspx

MANEJO DE EXCEPCIONES

Estructura TRY/CATCH

Una mejora importante que tenemos en SQL Server es el manejo de errores que ahora es posible en T-SQL con los bloques TRY/CATCH, sin olvidar la sintaxis que utilizamos para las transacciones.

Sintaxis:

```
BEGIN TRY
   BEGIN TRANSACTION;

   -- Bloque de código SQL a proteger

   COMMIT TRANSACTION;
END TRY
BEGIN CATCH
   ROLLBACK TRANSACTION;
   -- Código para mostrar el mensaje de la excepción
END CATCH;
```

Si ocurre un error dentro de la transacción en un bloque TRY inmediatamente se dirige al bloque CATCH.

Para poder acceder a la información del error tenemos las siguientes funciones:

ERROR_NUMBER()	:	Devuelve el número de error.
ERROR_MESSAGE()	:	Devuelve el texto completo del mensaje de error. El texto incluye los valores suministrados para los parámetros sustituibles, como longitudes, nombres de objeto u horas.
ERROR_SEVERITY()	:	Devuelve la gravedad del error.
ERROR_STATE()	:	Devuelve el número de estado de error.
ERROR_LINE()	:	Devuelve el número de línea donde se produjo el error.
ERROR_PROCEDURE()	:	Devuelve el nombre del procedimiento donde se produjo el error.

Más información en: http://technet.microsoft.com/es-pe/library/ms175976.aspx

Pasemos a crear un ejemplo bastante sencillo en donde vamos a provocar una excepción en una división por cero para observar el resultado de estas funciones.

Ejemplo 12: Captura de Error

```
BEGIN TRY
   DECLARE @TOTAL INT;
   SET @TOTAL = 20;
   SELECT @TOTAL/0
END TRY
BEGIN CATCH
   SELECT
      ERROR_NUMBER() AS Numero_de_Error,
      ERROR_SEVERITY() AS Gravedad_del_Error,
      ERROR_STATE() AS Estado_del_Error,
      ERROR_PROCEDURE() AS Procedimiento_del_Error,
      ERROR_LINE() AS Linea_de_Error,
      ERROR_MESSAGE() AS Mensaje_de_Error;
END CATCH;
GO
```

Un bloque CATCH no controla los siguientes tipos de errores cuando se producen en el mismo nivel de ejecución que la construcción TRY…CATCH:

- Errores de compilación, como errores de sintaxis, que impiden la ejecución de un lote.
- Errores que se producen durante la recompilación de instrucciones, como errores de resolución de nombres de objeto que se producen después de la compilación debido a una resolución de nombres diferida.

Estos errores se devuelven al nivel de ejecución del lote, procedimiento almacenado o desencadenador.

En el Ejemplo 13 se muestra cómo la estructura TRY…CATCH no captura un error de resolución de nombre de objeto generado por una instrucción SELECT, sin embargo, el bloque CATCH si captura el error cuando la misma instrucción SELECT se ejecuta dentro de un procedimiento almacenado, tal como se puede apreciar en el Ejemplo 14.

Ejemplo 13: Error no capturado

```
BEGIN TRY
   SELECT * FROM TablaNoExiste;
END TRY
BEGIN CATCH
   SELECT ERROR_NUMBER() AS ErrorNumber,
      ERROR_MESSAGE() AS ErrorMessage;
END CATCH;
GO

Mens. 208, Nivel 16, Estado 1, Línea 2
El nombre de objeto 'TablaNoExiste' no es válido.
```

El error no se captura y el control se transfiere hacia fuera de la estructura TRY…CATCH, al siguiente nivel superior.

Al ejecutar la instrucción SELECT dentro de un procedimiento almacenado, el error se produce en un nivel inferior al bloque TRY. La estructura TRY…CATCH controlará el error.

Ejemplo 14: Captura de error de un nivel inferior

```
USE EDUCA;
GO

IF OBJECT_ID ( N'usp_ProcEjemplo', N'P' ) IS NOT NULL
   DROP PROCEDURE usp_ProcEjemplo;
GO

CREATE PROCEDURE usp_ProcEjemplo
AS
   SELECT * FROM TablaNoExiste;
GO

BEGIN TRY
   EXECUTE usp_ProcEjemplo;
END TRY
BEGIN CATCH
   SELECT ERROR_NUMBER() AS ErrorNumber,
   ERROR_MESSAGE() AS ErrorMessage;
END CATCH;
GO

ErrorNumber ErrorMessage
--------- -------------------------------------------------------------
208         El nombre de objeto 'TablaNoExiste' no es válido.

(1 filas afectadas)
```

Más información en: http://technet.microsoft.com/es-pe/library/ms175976.aspx

Sentencia: THROW

Genera una excepción y transfiere el control a un bloque CATCH de una estructura TRY...CATCH.

Sintaxis

```
THROW [ { error_number | @local_variable },
        { message | @local_variable },
    { state | @local_variable }
] [ ; ]
```

Ejemplo 15: Generar una nueva excepción

```
THROW 51000, 'El registro no existe.', 1;
```

Ejemplo 16: Propagar la última excepción

```
BEGIN TRY

   -- Bloque a controlar

END TRY
BEGIN CATCH

   -- Proceso de control

   THROW; -- Propaga la última excepción
END CATCH;
```

Más información en: http://technet.microsoft.com/es-pe/library/ee677615.aspx

REQUERIMIENTOS A RESOLVER

Requerimiento 1

Base de Datos: **EDUTEC**

Se necesita un procedimiento que permita generar un nuevo ciclo, las condiciones son las siguientes:

1. Por cada año existen doce ciclos, uno por mes, por ejemplo: 2020-01, 2020-02, 2020-03, etc.
2. El procedimiento debe leer el último ciclo y generar el que continua.
3. La fecha de inicio es el primer día del mes.
4. La fecha de fin es el último día del mes.
5. El nuevo ciclo se debe grabar en la tabla CICLO.

Requerimiento 2

Base de Datos: **EDUTEC**

Se necesita un procedimiento para registrar una nueva matricula, las condiciones son las siguientes:

1. Se debe verificar que el curso programado exista y se encuentre vigente.
2. Se debe verificar que el alumno exista.
3. Se debe verificar que el alumno no se encuentre matriculado.
4. Se debe verificar que existan vacantes disponibles.
5. La matrícula se registra en la tabla MATRICULA.
6. Se deben actualizar las columnas vacantes y matriculados en la tabla CURSOPROGRAMADO.

Requerimiento 3

Base de datos: **EDUCA**

Se necesita un procedimiento para registrar un nuevo pago, las condiciones son las siguientes:

1. Se debe verificar que exista la matricula.
2. Se debe verificar que exista pago pendiente.
3. Si el pago es en una cuota, debe ser por el total del costo del curso.
4. Si el pago es la primera cuota, debe ser mínimo por el 30% del costo del curso.
5. Si se trata de la última cuota, debe ser por el saldo pendiente.
6. Si el pago es por el saldo pendiente, y es necesario corregir el número de cuotas se debe realizar.
7. En otros casos se debe aceptar el pago de la cuota.

Capítulo 7
TRABAJANDO CON CURSORES

TRABAJANDO CON CURSORES

Declaración

Sintaxis ISO

```
DECLARE cursor_name [ INSENSITIVE ] [ SCROLL ] CURSOR
   FOR select_statement
   [ FOR { READ ONLY | UPDATE [ OF column_name [ ,...n ] ] } ] [;]
```

Sintaxis Transact-SQL Extended

```
DECLARE cursor_name CURSOR [ LOCAL | GLOBAL ]
   [ FORWARD_ONLY | SCROLL ]
   [ STATIC | KEYSET | DYNAMIC | FAST_FORWARD ]
   [ READ_ONLY | SCROLL_LOCKS | OPTIMISTIC ]
   [ TYPE_WARNING ]
   FOR select_statement
   [ FOR UPDATE [ OF column_name [ ,...n ] ] ] [;]
```

Ejemplo 17

```
DECLARE cur_cursos CURSOR
FOR
   SELECT cur_id, cur_nombre, cur_precio
   FROM dbo.CURSO;
```

Abrir un cursor

Sintaxis

```
OPEN { { [ GLOBAL ] cursor_name } | cursor_variable_name }
```

Ejemplo 18

```
OPEN cur_cursos;
```

Recuperar filas de un cursor

Sintaxis

```
FETCH
   [ [ NEXT | PRIOR | FIRST | LAST
      | ABSOLUTE { n | @nvar } | RELATIVE { n | @nvar } ]
   FROM ]
{ { [ GLOBAL ] cursor_name } | @cursor_variable_name }
[ INTO @variable_name [ ,...n ] ]
```

Ejemplo 19

```
DECLARE @cur_id int, @cur_nombre varchar(100), @cur_precio money;

FETCH NEXT FROM cur_cursos
INTO @cur_id, @cur_nombre, @cur_precio;

SELECT @cur_id, @cur_nombre, @cur_precio;
```

Cerrar un cursor

Sintaxis

```
CLOSE { { [ GLOBAL ] cursor_name } | cursor_variable_name }
```

Ejemplo 20

```
CLOSE cur_cursos;
```

Liberar recursos de un cursor

Sintaxis

```
DEALLOCATE { { [ GLOBAL ] cursor_name } | @cursor_variable_name }
```

Ejemplo 21

```
DEALLOCATE cur_cursos;
```

Ejemplo 22: Ejemplo completo

```
DECLARE cur_cursos CURSOR
FOR
   SELECT cur_id, cur_nombre, cur_precio
   FROM dbo.CURSO;

OPEN cur_cursos;

DECLARE @cur_id int, @cur_nombre varchar(100), @cur_precio money;

FETCH NEXT FROM cur_cursos
INTO @cur_id, @cur_nombre, @cur_precio;

SELECT @cur_id, @cur_nombre, @cur_precio;

CLOSE cur_cursos;

DEALLOCATE cur_cursos;
```

CONTROL DE UN CURSOR

Variable: @@FETCH_STATUS

Devuelve el estado de la última instrucción FETCH emitida para cualquier cursor abierto en ese momento por la conexión.

Sintaxis

```
@@FETCH_STATUS
```

Valor devuelto

Valor devuelto	Descripción
0	La instrucción FETCH se ejecutó correctamente.
-1	La instrucción FETCH no se ejecutó correctamente o la fila estaba más allá del conjunto de resultados.
-2	Falta la fila capturada.

Ejemplo 23

```
DECLARE cur_cursos CURSOR
FOR
   SELECT cur_id, cur_nombre, cur_precio
   FROM dbo.CURSO;

OPEN cur_cursos;

DECLARE @cur_id int, @cur_nombre varchar(100), @cur_precio money

FETCH NEXT FROM cur_cursos
INTO @cur_id, @cur_nombre, @cur_precio;

WHILE( @@FETCH_STATUS = 0 )
BEGIN
   PRINT CONCAT(@cur_id, ' - ', @cur_nombre, ' - ', @cur_precio);
   FETCH NEXT FROM cur_cursos
   INTO @cur_id, @cur_nombre, @cur_precio;
END;

CLOSE cur_cursos;

DEALLOCATE cur_cursos;
```

Variable: @@CURSOR_ROWS

Devuelve el número de filas certificadas que se encuentran en el último cursor abierto en la conexión actual. Para mejorar el rendimiento, SQL Server puede rellenar asincrónicamente los cursores estáticos y de conjunto de claves de gran tamaño. Puede llamar a @@CURSOR_ROWS para determinar que el número de filas que cumplan las condiciones del cursor se recuperen en el momento en que se llama a @@CURSOR_ROWS.

Sintaxis

```
@@CURSOR_ROWS
```

Valor devuelto

Valor devuelto	Descripción
-m	El cursor se rellena de forma asincrónica. El valor devuelto (-m) es el número de filas que el conjunto de claves contiene actualmente.
-1	El cursor es dinámico. Como los cursores dinámicos reflejan todos los cambios, el número de filas correspondientes al cursor cambia constantemente. Nunca se puede afirmar que se han recuperado todas las filas que correspondan.
0	No se han abierto cursores, no hay filas calificadas para el último cursor abierto, o éste se ha cerrado o su asignación se ha cancelado.
n	El cursor está completamente relleno. El valor devuelto (n) es el número total de filas del cursor.

Ejemplo 24

El este ejemplo se declara un cursor y se utiliza la sentencia PRINT para mostrar el valor de @@CURSOR_ROWS.

@@CURSOR_ROWS, tiene el valor 0 antes de abrir el cursor y el valor -1 después de abrir el cursor, lo que indica que el número de filas es dinámico, cambia constantemente.

```
DECLARE cur_cursos CURSOR
FOR
   SELECT cur_id, cur_nombre, cur_precio
   FROM dbo.CURSO;

PRINT CONCAT('CURSOR_ROWS = ', @@CURSOR_ROWS);

OPEN cur_cursos;

PRINT CONCAT('CURSOR_ROWS = ', @@CURSOR_ROWS);

CLOSE cur_cursos;

DEALLOCATE cur_cursos;
GO

CURSOR_ROWS = 0
CURSOR_ROWS = -1
```

Función: CURSOR_STATUS ( )

Una función escalar que permite a quien llama a un procedimiento almacenado determinar si el procedimiento ha devuelto un cursor y el conjunto de resultados de un parámetro determinado.

Sintaxis

```
CURSOR_STATUS
   (
      { 'local' , 'cursor_name' }
      { 'global' , 'cursor_name' }
      | { 'variable' , 'cursor_variable' }
   )
```

Valor devuelto

Valor devuelto	Nombre de cursor	Variable de cursor
1	El conjunto de resultados del cursor tiene al menos una fila. Para los cursores INSENSITIVE y de conjunto de claves, el conjunto de resultados tiene al menos una fila. Para los cursores dinámicos, el conjunto de resultados puede tener cero, una o más filas.	El cursor asignado a esta variable está abierto. Para los cursores INSENSITIVE y de conjunto de claves, el conjunto de resultados tiene al menos una fila. Para los cursores dinámicos, el conjunto de resultados puede tener cero, una o más filas.
0	El conjunto de resultados del cursor está vacío.*	El cursor asignado a esta variable está abierto, pero el conjunto de resultados está definitivamente vacío.*
-1	El cursor está cerrado.	El cursor asignado a esta variable está cerrado.

-2	No aplicable.	Puede ser: El procedimiento llamado anteriormente no ha asignado ningún cursor a esta variable OUTPUT. El procedimiento llamado anteriormente asignó un cursor a esta variable OUTPUT, pero se encontraba en un estado cerrado al terminar el procedimiento. Por tanto, se cancela la asignación del cursor y no se devuelve al procedimiento que hace la llamada. No hay ningún cursor asignado a una variable declarada de cursor.
-3	No existe ningún cursor con el nombre indicado.	No existe una variable de cursor con el nombre indicado o, si existe, no tiene todavía ningún cursor asignado.

Ejemplo 25

El siguiente script muestra el estado del cursor antes y después de la apertura.

```
-- Seleccionando la base de datos

USE EduTec;
GO

-- Crea un cursor

DECLARE cur_demo CURSOR
FOR SELECT * FROM dbo.Curso;

-- Cursor cerrado

SELECT CURSOR_STATUS('global','cur_demo') AS 'Después de declarar';

-- Cursor abierto

OPEN cur_demo;
SELECT CURSOR_STATUS('global','cur_demo') AS 'Después de abrir';

-- Cursor cerrado

CLOSE cur_demo;
SELECT CURSOR_STATUS('global','cur_demo') AS 'Después de cerrar';

-- Remover el cursor

DEALLOCATE cur_demo;
GO
```

Se obtiene el siguiente resultado:

```
Después de declarar
-------------------
-1

(1 filas afectadas)

Después de abrir
----------------
1

(1 filas afectadas)

Después de cerrar
-----------------
-1

(1 filas afectadas)
```

BUCLE DE EXTRACCIÓN

Plantilla

```
FETCH NEXT FROM <nombre_cursor> INTO <lista_variables>;
WHILE ( @@FETCH_STATUS = 0 )
BEGIN

   --   Proceso

   FETCH NEXT FROM <nombre_cursor> INTO <lista_variables>;
END
```

Ejemplo 26

El siguiente script muestra la cantidad de empleados que hay en cada departamento:

```
USE RH;
GO

DECLARE cur_depts CURSOR
FOR
   SELECT
      d.iddepartamento codido,
      d.nombre nombre,
      count(*) emps
   FROM dbo.departamento d
   join dbo.empleado e
   on d.iddepartamento = e.iddepartamento
   group by d.iddepartamento, d.nombre;

OPEN cur_depts;

DECLARE @codigo int, @nombre varchar(100), @emps int

PRINT CONCAT('COD', SPACE(4),
     LEFT('NOMBRE' + SPACE(20), 20), SPACE(4), 'EMPS');
PRINT '-----------------------------------';

FETCH NEXT FROM cur_depts INTO @codigo, @nombre, @emps;
WHILE( @@FETCH_STATUS = 0 )
BEGIN
   PRINT CONCAT(@codigo, SPACE(4),
      LEFT(@nombre + SPACE(20), 20), SPACE(4), @emps);
   FETCH NEXT FROM cur_depts     INTO @codigo, @nombre, @emps;
```

```
END;

CLOSE cur_depts;
DEALLOCATE cur_depts;
GO
```

El resultado que se obtiene es similar al siguiente:

```
COD    NOMBRE                    EMPS
-----------------------------------
100    Gerencia                  2
101    Contabilidad              3
102    Investigacion             6
103    Ventas                    7
105    Sistemas                  4
```

EJERCICIOS

Ejercicio 12

En la base de datos RH, crear un procedimiento que de cada departamento muestre el trabajador con menor salario y el trabajador con mayor tiempo de servicio. Se deben mostrar los empates.

Ejercicio 13

En la base de EDUCA, crear un procedimiento que muestre el alumno con mayor nota. Se deben mostrar los empates.

USO DE TABLAS TEMPORALES

Variables de tipo tabla

Las Variables de Tabla que son un tipo de datos que puede ser utilizados en un lote T-SQL (Batch), procedimiento almacenado o función; estas variables de tabla son creadas y definidas de forma similar a una tabla, sólo que tienen un alcance de vida bien definido. Las Variables de tabla suelen ser buenos reemplazos de tablas temporales siempre y cuando el conjunto de datos es pequeño.

Razones para usar las variables de tabla:

- **Duración o alcance.** La duración de la variable de tabla sólo vive durante la ejecución del lote, función, o procedimiento almacenado.
- **Tiempos de bloqueo más cortos.** Por el estrecho alcance o tiempo de vida.
- **Menos re compilaciones.** Cuando se usa en los procedimientos almacenados.

El inconveniente de utilizar las variables de tabla es su rendimiento. El rendimiento de las variables de tabla se ve afectado cuando el resultado es demasiado grande o cuando los datos de la columna de cardinalidad son fundamentales para la optimización del proceso de consulta.

La sintaxis para crear una variable de tabla es similar a la de crear una tabla normal, se utiliza la palabra clave DECLARE y el nombre de tabla, anteponiendo el símbolo @.

Sintaxis:

```
DECLARE @TableName TABLE
(
column_name <data_type> [ NULL | NOT NULL ]
[ ,...n ]
)
```

Ejemplo 27

En el siguiente script se crea una variable tipo TABLE, se inserta valores y finalmente se consulta.

```
-- Creando la variable de tipo tabla.
DECLARE @Catalogo TABLE
(
   idProd int NOT NULL PRIMARY KEY,
   nombre varchar(30) NULL,
   precio money
);

-- Insertando datos en la variable de tipo tabla.
INSERT INTO @Catalogo VALUES
(1, 'Refrigeradora', 2400.0),
(2, 'Televisor', 3500.0),
(3, 'Laptop', 4500.0);

-- Consultando datos de la variable de tipo tabla.
SELECT * FROM @Catalogo;
GO
```

El resultado que se obtiene es el siguiente:

```
idProd      nombre                         precio
----------- ------------------------------ ---------------------
1           Refrigeradora                  2400.00
2           Televisor                      3500.00
3           Laptop                         4500.00

(3 filas afectadas)
```

Al terminar la ejecución del batch o bloque de instrucciones automáticamente se eliminará la variable tabla.

Tablas temporales locales

Las tablas temporales locales están disponibles para usarse en la sesión actual. Varias conexiones pueden crear una tabla temporal con mismo nombre, esto solo para tablas temporales locales, sin causar conflictos. La representación interna de la tabla local tiene un nombre único, para no estar en conflicto con otras tablas temporales con el mismo nombre creado por otras conexiones en la base de datos tempdb.

Las tablas temporales locales son eliminadas con el comando DROP o se eliminan automáticamente de memoria cuando se cierra la conexión del usuario.

Las tablas temporales locales se crean anteponiendo el símbolo # al nombre de la tabla.

Ejemplo 28

Creación de la tabla temporal.

```
-- Creación de la tabla temporal
CREATE TABLE #ResumenVentas
(
   idProd int NOT NULL PRIMARY KEY,
   nombre varchar(30) NULL,
   ventas money NULL
);
GO
```

Insertando datos en la tabla temporal, desde la misma sesión.

```
-- Insertando datos
INSERT INTO #ResumenVentas VALUES
(1, 'Refrigeradora', 45657.0),
(2, 'Televisor', 65350.0),
(3, 'Laptop', 145640.0);
GO
```

Consultando la tabla temporal.

```
-- Consultando la tabla temporal
SELECT *  FROM #ResumenVentas;
```

Resultado obtenido.

```
idProd      nombre                         ventas
----------- ------------------------------ ---------------------
1           Refrigeradora                  45657.00
2           Televisor                      65350.00
3           Laptop                         145640.00

(3 filas afectadas)
```

Eliminado la tabla temporal.

```
-- Eliminando la tabla
DROP TABLE #ResumenVentas;
GO
```

Para que el ejemplo funcione las instrucciones deben ejecutarse en la misma sesión.

Tablas temporales globales

Las tablas temporales globales tienen un alcance diferente al de las tablas temporales locales. Una vez que se crea una tabla temporal global en una sesión, cualquier usuario con permisos adecuados sobre la base de datos puede acceder a la tabla desde cualquier otra sesión. A diferencia de tablas temporales locales, no se pueden crear versiones simultáneas de una tabla temporal global, ya que esto generará un conflicto de nombres.

Las tablas temporales globales de eliminan explícitamente de SQL Server ejecutando DROP TABLE. También se eliminan automáticamente después de que se cierra la sesión que la crea, la tabla temporal global no es referenciada por otras conexiones, pero es muy raro ver que se utilicen tablas temporales globales en bases de datos en producción.

Es importante considerar cuando una tabla va o debe ser compartida a través de conexiones, se debe crear una tabla real, en lugar de una tabla temporal global. No obstante, SQL Server ofrece esto como una opción.

Las tablas temporales globales se crean anteponiendo el símbolo ## al nombre de la tabla.

Ejemplo 29

Creando la tabla temporal global.

```
--Creando la tabla temporal global
CREATE TABLE ##Autores
( id int NOT NULL IDENTITY(1,1) PRIMARY KEY,
  nombre varchar(30) NULL
);
GO
```

Insertar datos en la tabla temporal.

```
-- Insertando datos
INSERT INTO ##Autores(nombre) VALUES
('Gustavo Coronel'),
('Sergio Matsukawa'),
('Ricardo Marcelo');
GO
```

La consulta a esta tabla se puede hacer desde cualquier otra sesión.

```
-- Consultando la tabla temporal
select * from ##Autores;
GO
```

El resultado es el siguiente.

```
id          nombre
----------- ------------------------------
1           Gustavo Coronel
2           Sergio Matsukawa
3           Ricardo Marcelo

(3 filas afectadas)
```

EJERCICIOS

Ejercicio 14

En la base de datos RH, crear un procedimiento que de cada departamento muestre lo siguiente:

- La cantidad de empleados
- Planilla sin comisión
- Planilla con comisión

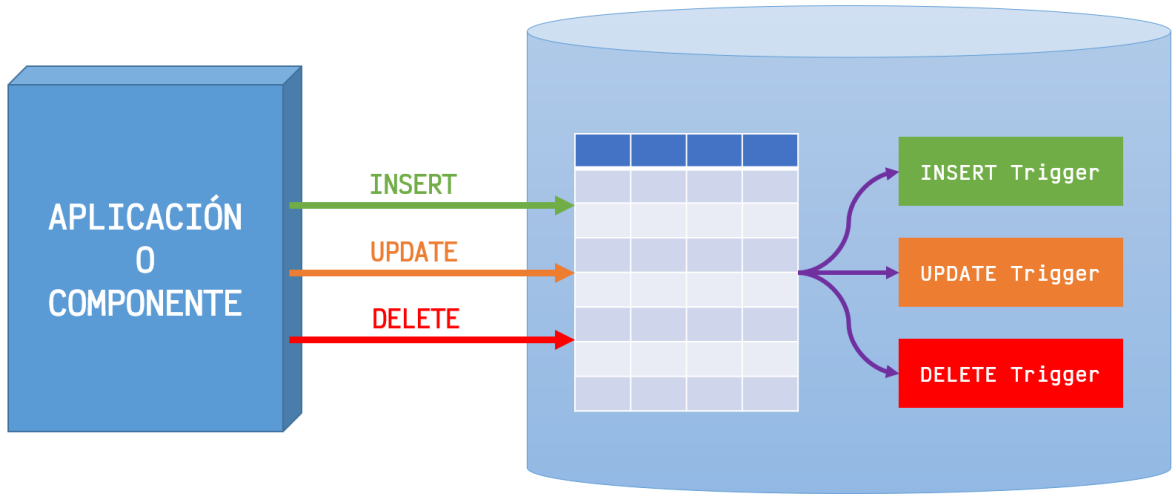
Ejercicio 15

En la base de datos EDUTEC, crear un procedimiento que reciba como parámetro un periodo, por ejemplo 2010, 2011, 2012, etc. y reporte por cada ciclo en el periodo los siguientes datos:

- Cantidad de cursos programados
- Cantidad de alumnos proyectados
- Cantidad de alumnos matriculados
- Importe proyectado (de alumnos proyectados)
- Importe real (de alumnos matriculados)

Capítulo 8
GESTIÓN DE TRIGGERS

INTRODUCCIÓN



Los triggers DDL se pueden usar para auditar las operaciones de base de datos o de servidor que crean, modifican o eliminan objetos de base de datos o garantizar que las instrucciones DDL aplican las reglas de negocios antes de que se ejecuten.

Los triggers DDL se inician en respuesta a una variedad de eventos de lenguaje de definición de datos (DDL). Estos eventos corresponden principalmente a las instrucciones Transact SQL que comienzan por las palabras clave CREATE, ALTER, DROP, GRANT, DENY, REVOKE o UPDATE STATISTICS. Algunos procedimientos almacenados del sistema que ejecutan operaciones de tipo DDL también pueden activar triggers DDL.

Importante

Los triggers DDL se pueden utilizar para verificar la respuesta a los procedimientos almacenados del sistema. Por ejemplo, la sentencia CREATE TYPE y el procedimiento almacenado SP_ADDTYPE activarán un trigger DDL activado por el evento CREATE_TYPE.

Cuando usar triggers DDL:

- Para evitar determinados cambios en el esquema de la base de datos.
- Se quiere que ocurra algún evento en la base de datos como respuesta a un cambio realizado en el esquema de la base de datos.
- Se quiere registrar cambios o eventos del esquema de la base de datos.

Para mayor información sobre los eventos DDL consultar la siguiente dirección:

http://msdn.microsoft.com/es-es/library/bb522542.aspx

TIPOS DE TRIGGERS DDL

Trigger Transact-SQL DDL

Se trata de un tipo especial de procedimiento almacenado de Transact-SQL que ejecuta una o más instrucciones Transact-SQL en respuesta a un evento de servidor o de base de datos.

Por ejemplo, un trigger DDL se puede activar si se ejecuta una instrucción como ALTER SERVER CONFIGURATION o si se elimina una tabla mediante DROP TABLE.

Desencadenante CLR DLL

En lugar de ejecutar un procedimiento almacenado Transact-SQL, un trigger CLR ejecuta uno o más métodos escritos en código administrado que son miembros de un ensamblado creado en .NET Framework y cargado en SQL Server.

Los triggers DDL solo se activan cuando se ejecutan las instrucciones DDL que los desencadenan. Los triggers DDL no se pueden usar como triggers INSTEAD OF. Los triggers DDL no se activan como respuesta a eventos que afectan a procedimientos almacenados y tablas temporales, ya sean locales o globales.

Los triggers DDL no crean las tablas especiales **inserted** y **deleted**.

La información acerca de un evento que activa un trigger DDL y las modificaciones posteriores provocadas por el mismo se capturan con la función EVENTDATA.

Para cada evento DDL se pueden crear varios triggers.

A diferencia de los triggers DML, los triggers DDL no tienen como ámbito los esquemas. Por tanto, las funciones como OBJECT_ID, OBJECT_NAME, OBJECTPROPERTY y OBJECTPROPERTYEX no se pueden usar para efectuar consultas en metadatos sobre triggers DDL. En su lugar se debe usar las vistas de catálogo.

Los triggers DDL con ámbito en el servidor aparecen en el Explorador de objetos de SQL Server Management Studio, en la carpeta **Triggers**. Dicha carpeta se encuentra en la carpeta **Server Objects**. Los triggers DDL de ámbito de base de datos aparecen en la carpeta **Database Triggers**. Esta carpeta se encuentra en la carpeta **Programmability** de la base de datos correspondiente.

Ámbito de los triggers DDL

Los triggers DDL pueden activarse en respuesta a un evento de Transact-SQL procesado en la base de datos actual o en el servidor actual. El ámbito del desencadenante depende del evento.

Por ejemplo, un trigger DDL creado para activarse como respuesta al evento CREATE_TABLE se activará siempre que se produzca un evento CREATE_TABLE en la base de datos o en la instancia de servidor. Un trigger DDL creado para activarse como respuesta a un evento CREATE_LOGIN se activará únicamente cuando se produzca un evento CREATE_LOGIN en la instancia de servidor.

Ejemplo 30

En el siguiente ejemplo, el trigger DDL TR_SEGURIDAD se activará siempre que se produzca un evento DROP_TABLE o ALTER_TABLE en la base de datos, impidiendo que se elimine o modifique una tabla.

```
USE EDUCA;
GO

IF EXISTS ( SELECT 1 FROM sys.triggers WHERE name = 'tr_seguridad')
BEGIN
DROP TRIGGER tr_seguridad ON DATABASE;
END;
GO

CREATE TRIGGER tr_seguridad
ON DATABASE
FOR DROP_TABLE, ALTER_TABLE
AS
BEGIN
RAISERROR('No se permite eliminar ni modificar tablas.',16,1);
ROLLBACK;
END;
GO

DROP TABLE dbo.PAGO;
GO
```

Se obtiene el siguiente mensaje:

```
Mens 50000, Nivel 16, Estado 1, Procedimiento tr_seguridad, Línea 7
No se permite eliminar ni modificar tablas.
```

```
Mens. 3609, Nivel 16, Estado 2, Línea 2
La transacción terminó en el desencadenador. Se anuló el lote.
```

Eliminemos el desencadenante:

```
DROP TRIGGER tr_seguridad ON DATABASE;
GO
```

Ejemplo 31

En el siguiente ejemplo, un trigger DDL imprime un mensaje si se produce algún evento CREATE_DATABASE en la instancia de servidor actual. El ejemplo usa la función EVENTDATA para recuperar el texto de la instrucción Transact-SQL correspondiente.

```
USE master;
GO

IF EXISTS (SELECT * FROM sys.server_triggers WHERE name = 'tr_ddl_database')
BEGIN
   DROP TRIGGER tr_ddl_database ON ALL SERVER;
END;
GO

CREATE TRIGGER tr_ddl_database
ON ALL SERVER
FOR CREATE_DATABASE
AS
BEGIN
   DECLARE  @data XML, @sql varchar(2000);
   set @data = EVENTDATA();
   set @sql = @data.value('(/EVENT_INSTANCE/TSQLCommand)[1]', 'varchar(2000)');
   PRINT 'Base de datos creada.'
   PRINT CONCAT('Sentencia: ', @sql);
END;
GO

CREATE DATABASE DEMO;
GO
```

Se obtiene el siguiente mensaje:

```
Base de datos creada.
```

```
Sentencia: CREATE DATABASE DEMO4;
```

Eliminemos el desencadenante:

```
DROP TRIGGER tr_ddl_database ON ALL SERVER;
GO
```

Los triggers DDL de base de datos se almacenan como objetos en la base de datos donde se crean. Se puede utilizar la vista de catálogo **sys.triggers** para obtener información de estos triggers.

Los triggers DDL del servidor se almacenan como objetos en la base de datos master. Sin embargo, puede obtenerse información sobre los triggers DDL de servidor si se consulta la vista de catálogo **sys.server_triggers** en cualquier contexto de base de datos.

MANTENIMIENTO DE TRIGGERS DDL

Creación de trigger DDL

Los triggers DDL se crean con la instrucción CREATE TRIGGER de Transact-SQL para desencadenadores DDL.

Para crear un trigger:

```
CREATE TRIGGER . . .
```

Modificar triggers DDL

Si necesita modificar la definición de un trigger DDL, puede quitarlo y volver a crear, o bien volver a definirlo en un solo paso.

Si cambia el nombre de un objeto al que hace referencia un trigger DDL, debe modificar el trigger para que el texto refleje el nuevo nombre. Por lo tanto, antes de cambiar el nombre de un objeto, vea primero las dependencias del mismo para determinar si algún trigger va a verse afectado por el cambio propuesto.

Para modificar un desencadenante:

```
ALTER TRIGGER . . .
```

Para ver las dependencias de un desencadenante:

- sys.sql_expression_dependencies
- sys.dm_sql_referenced_entities
- sys.dm_sql_referencing_entities

Deshabilitar y eliminar triggers DDL

Se debe eliminar o deshabilitar un desencadenante DDL cuando ya no se necesite.

Al deshabilitar un trigger DDL, éste no se elimina de la base de datos. Sigue siendo un objeto de la base de datos actual. Sin embargo, el trigger no se activa cuando se ejecuta una instrucción Transact-SQL en la que se programó. Los triggers DDL deshabilitados se pueden volver a habilitar. La habilitación de un trigger DDL hace que se active tal como lo hizo cuando se creó originalmente. Cuando se crean triggers DDL, se habilitan de forma predeterminada.

Cuando el trigger DDL se elimina, se quita de la base de datos actual. Los objetos o datos incluidos en el ámbito del trigger DDL no se ven afectados.

Para deshabilitar un trigger DDL tenemos:

- DISABLE TRIGGER
- ALTER TABLE

Para habilitar un trigger DDL:

- ENABLE TRIGGER
- ALTER TABLE

Para eliminar un trigger DDL:

- DROP TRIGGER

EJEMPLOS

Ejemplo 32: Log de cambios en el sistema

El siguiente trigger crea los registrar de los cambios que se realizan en la base de datos EDUTEC.

Creación de la base de datos de auditoría.

```
-- Creación de la base de datos
CREATE DATABASE AuditDB;
GO

-- Seleccionamos la base de datos
USE AuditDB;
GO

-- Tabla de auditoria.
CREATE TABLE dbo.DDLEvents
(
   EventDate    DATETIME NOT NULL DEFAULT CURRENT_TIMESTAMP,
   EventType    NVARCHAR(64),
   EventDDL     NVARCHAR(MAX),
   EventXML     XML,
   DatabaseName NVARCHAR(255),
   SchemaName   NVARCHAR(255),
   ObjectName   NVARCHAR(255),
   HostName     VARCHAR(64),
   IPAddress    VARCHAR(32),
   ProgramName  NVARCHAR(255),
   LoginName    NVARCHAR(255)
);
```

Creación del desencadenante en la base de datos EDUTEC.

```
-- Seleccionar la Base de Datos
USE EDUTEC;
GO

-- Creación del desencadenante
CREATE TRIGGER DDL_Change_Log
    ON DATABASE
    FOR CREATE_PROCEDURE, ALTER_PROCEDURE, DROP_PROCEDURE,
        CREATE_TABLE, ALTER_TABLE, DROP_TABLE,
        CREATE_FUNCTION, ALTER_FUNCTION, DROP_FUNCTION
```

```
AS
BEGIN
   SET NOCOUNT ON;
   DECLARE
      @EventData XML = EVENTDATA(),
      @ip VARCHAR(32) =
         (
            SELECT client_net_address
            FROM sys.dm_exec_connections
            WHERE session_id = @@SPID
         );

   INSERT AuditDB.dbo.DDLEvents
   (
      EventType,
      EventDDL,
      EventXML,
      DatabaseName,
      SchemaName,
      ObjectName,
      HostName,
      IPAddress,
      ProgramName,
      LoginName
    )
   SELECT
      @EventData.value('(/EVENT_INSTANCE/EventType)[1]',   'NVARCHAR(100)'),
      @EventData.value('(/EVENT_INSTANCE/TSQLCommand)[1]', 'NVARCHAR(MAX)'),
      @EventData,
      DB_NAME(),
      @EventData.value('(/EVENT_INSTANCE/SchemaName)[1]',  'NVARCHAR(255)'),
      @EventData.value('(/EVENT_INSTANCE/ObjectName)[1]',  'NVARCHAR(255)'),
      HOST_NAME(),
      @ip,
      PROGRAM_NAME(),
      SUSER_SNAME();
END;
GO
```

Provocando el disparo del desencadenante.

```
-- Seleccionando la base de datos
USE EDUTEC;
GO

-- Creando un procedimiento almacenado
CREATE PROC dbo.usp_GetDBVersion
AS
SELECT @@version AS Vrsion;
GO
```

Consultando el log.

```
SELECT * FROM AuditDB.dbo.DDLEvents;
GO
```

Ejemplo 33: Log de cambios de inicios de sesión u usuarios

Creando la tabla en la base de datos master.

```
-- Creando la tabla

USE master;
GO

CREATE TABLE DDLSecurityLog
(
   DDLSecurityLog_ID int IDENTITY(1, 1) NOT NULL,
   InsertionDate datetime NOT NULL
      CONSTRAINT DF_ddl_log_InsertionDate
      DEFAULT ( GETDATE() ),
   CurrentUser nvarchar(50) NOT NULL
      CONSTRAINT DF_ddl_log_CurrentUser
      DEFAULT ( CONVERT(nvarchar(50), USER_NAME(), 0 ) ),
   LoginName nvarchar(50) NOT NULL
      CONSTRAINT DF_DDLSecurityLog_LoginName
      DEFAULT ( CONVERT(nvarchar(50), SUSER_SNAME(), ( 0 )) ),
   Username nvarchar(50) NOT NULL
      CONSTRAINT DF_DDLSecurityLog_Username
      DEFAULT ( CONVERT(nvarchar(50), original_login(), ( 0 )) ),
   EventType nvarchar(100) NULL,
   objectName nvarchar(100) NULL,
   objectType nvarchar(100) NULL,
```

```
   DatabaseName nvarchar(100) NULL,
   tsql nvarchar(MAX) NULL
) ;
GO
```

Creando el trigger de seguridad en el servidor.

```
-- Creando el trigger de seguridad para el servidor

USE master;
GO

IF EXISTS ( SELECT  *
            FROM    sys.server_triggers
            WHERE   name = 'trgLogServerSecurityEvents' )
    DROP TRIGGER trgLogServerSecurityEvents ON ALL SERVER;
GO


CREATE TRIGGER trgLogServerSecurityEvents ON ALL SERVER
    FOR CREATE_LOGIN, ALTER_LOGIN, DROP_LOGIN, GRANT_SERVER, DENY_SERVER,
        REVOKE_SERVER, ALTER_AUTHORIZATION_SERVER
AS
BEGIN
   DECLARE @data XML
   SET @data = EVENTDATA()
   INSERT  INTO master..DDLSecurityLog
   (
      EventType,
      ObjectName,
      ObjectType,
      DatabaseName,
      tsql
   )
   VALUES
   (
      @data.value('(/EVENT_INSTANCE/EventType)[1]', 'nvarchar(100)'),
      @data.value('(/EVENT_INSTANCE/ObjectName)[1]', 'nvarchar(100)'),
      @data.value('(/EVENT_INSTANCE/ObjectType)[1]', 'nvarchar(100)'),
      'Server',
      @data.value('(/EVENT_INSTANCE/TSQLCommand)[1]', 'nvarchar(max)')
   );
END;
GO
```

Trigger de seguridad en cada base de datos.

```
USE RH;
GO

CREATE TRIGGER trgLogDatabaseSecurityEvents
ON DATABASE
FOR DDL_DATABASE_SECURITY_EVENTS
AS
BEGIN
   DECLARE @data XML;
   SET @data = EVENTDATA();
   INSERT  INTO master..DDLSecurityLog
   (
      EventType,
      ObjectName,
      ObjectType,
      DatabaseName,
      tsql
   )
    VALUES
   (
      @data.value('(/EVENT_INSTANCE/EventType)[1]', 'nvarchar(100)'),
      @data.value('(/EVENT_INSTANCE/ObjectName)[1]', 'nvarchar(100)'),
      @data.value('(/EVENT_INSTANCE/ObjectType)[1]', 'nvarchar(100)'),
      @data.value('(/EVENT_INSTANCE/DatabaseName)[1]', 'nvarchar(max)'),
      @data.value('(/EVENT_INSTANCE/TSQLCommand)[1]', 'nvarchar(max)')
   );
END;
GO
```

Forzando el disparo de los trigger.

```
USE master;
GO

CREATE LOGIN intruso
   WITH
      PASSWORD= N'123456',
      DEFAULT_DATABASE = RH;
GO

EXEC master..sp_addsrvrolemember
   @loginame = N'intruso',
   @rolename = N'sysadmin';
GO

USE RH;
GO

CREATE USER intruso FOR LOGIN intruso
GO
```

Capítulo 9
PRACTICAS

PRACTICA 1

Problema 1

Desarrollar una función que permita encontrar el mayor de 5 números.

Problema 2

Desarrollar un procedimiento para calcular el promedio de un estudiante, se sabe que son 5 notas y se promedian las 4 mejores.

También se debe determinar la condición del estudiante, si el promedio es mayor o igual que 14 está aprobado, caso contrario esta desaprobado.

Problema 3

Desarrollar un procedimiento que permita calcular el sueldo neto de un trabajador según el número de horas trabajadas.

Si excede las 120 horas se le pagará 50% más por hora, solo por las horas que exceden las 120 horas.

Si el sueldo excede los 2000.0 Soles se le descuenta 8% por impuesto a la renta.

Problema 4

Desarrollar una función que lea el día y mes de nacimiento de una persona y determine a que signo pertenece.

PRACTICA 2

Problema 5

Desarrollar una función que permita convertir un número de base 10 a base 16.

Problema 6

Un padre con la intensión de motivar el estudio en su Hijo, le dice que será compensado según su promedio final del ciclo.

La compensación es monetaria, según el siguiente cuadro.

RANGO DE NOTA	FACTOR A MULTIPLICAR
0 - 5	50
6 - 12	80
13 - 17	120
18 - 20	500

Por ejemplo:

Si el promedio es: 13

La compensación es: 5x50 + 7x80 + 1x120 = 930

Desarrollar una función o procedimiento que permita calcular la compensación.

Problema 7

Desarrollar un procedimiento que permita mostrar la tabla de multiplicar de un número N. El valor de N se debe ingresar como un parámetro de entrada.

Problema 8

Desarrollar una función que permita calcular la potencia de un número, de tener 2 parámetros, uno para la base y otro para el exponente.

PRACTICA 3

Base de datos

La base de datos para esta práctica es EUREKABANK.

Problema 9

Desarrollar un procedimiento para registrar un nuevo cliente.

El procedimiento debe generar el código del cliente y retornarlo a través de un parámetro de salida.

Problema 10

Desarrollar un procedimiento para registrar un deposito en una cuenta especifica.

Problema 11

Desarrollar un procedimiento para registrar un retiro de una cuenta especifica.

Problema 12

Desarrollar un procedimiento para registrar una transferencia.

Se debe verificar que las cuentas origen y destino sean de la misma moneda.

PRACTICA 4

Problema 13

Base de datos: **EUREKABANK**

Desarrollar un procedimiento que reporte los saldos en SOLES y DOLARES de cada sucursal.

La plantilla del reporte es el siguiente:

CODIGO	NOMBRE	SALDO SOLES	SALDO DOLARES
001	Sipan	##,###.##	##,###.##
002	Chan Chan	##,###.##	##,###.##
003	Los Olivos	##,###.##	##,###.##
004	Pardo	##,###.##	##,###.##
005	Misti	##,###.##	##,###.##
006	Machupicchu	##,###.##	##,###.##
007	Grau	##,###.##	##,###.##

Problema 14

Base de datos: **EDUTEC**

Desarrollar un procedimiento que reporte el resumen de un determinado ciclo, según el siguiente formato:

CODIGO CURSO	NOMBRE CURSO	NUMERO SECCIONES	VACANTES PROGRAMADAS	MATRICULADOS	INGRESOS PROYECTADOS	INGRESOS REALES
A###	Aaaaaaaaaaa	###	###	###	##,###.##	##,###.##
A###	Aaaaaaaaaaa	###	###	###	##,###.##	##,###.##
A###	Aaaaaaaaaaa	###	###	###	##,###.##	##,###.##
A###	Aaaaaaaaaaa	###	###	###	##,###.##	##,###.##
A###	Aaaaaaaaaaa	###	###	###	##,###.##	##,###.##
A###	Aaaaaaaaaaa	###	###	###	##,###.##	##,###.##
A###	Aaaaaaaaaaa	###	###	###	##,###.##	##,###.##

Donde:

- Los **"INGRESOS PROYECTADOS"** se calcula en base a las vacantes programadas.
- Los **"INGRESOS REALES"** se calcula en base a la cantidad de matriculados.

Problema 15

Base de datos: **EDUTEC**

Desarrollar un procedimiento almacenado que permita obtener el pago que se debe hacer a cada uno de los profesores en un determinado ciclo.

El esquema del reporte es el siguiente:

CODIGO	NOMBRE	SECCIONES	HORAS	IMPORTE
P###	Aaaaaaaa	###	###	##,###.##
P###	Aaaaaaaa	###	###	##,###.##
P###	Aaaaaaaa	###	###	##,###.##
P###	Aaaaaaaa	###	###	##,###.##
P###	Aaaaaaaa	###	###	##,###.##
P###	Aaaaaaaa	###	###	##,###.##
P###	Aaaaaaaa	###	###	##,###.##

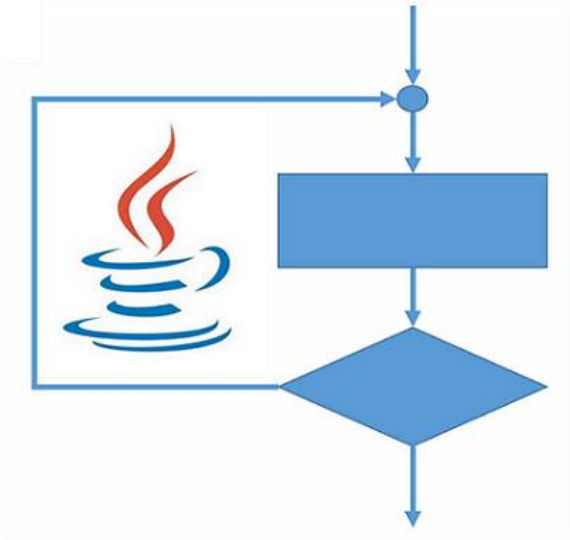
CURSOS VIRTUALES

CUPONES

En esta URL se publican cupones de descuento:

http://gcoronelc.github.io

FUNDAMENTOS DE PROGRAMACIÓN CON JAVA



Tener bases sólidas de programación muchas veces no es fácil, creo que es principalmente por que en algún momento de tu aprendizaje mezclas la entrada de datos con el proceso de los mismos, o mezclas el proceso con la salida o reporte, esto te lleva a utilizar malas prácticas de programación que luego te serán muy difíciles de superar.

En este curso aprenderás las mejores prácticas de programación para que te inicies con éxito en este competitivo mundo del desarrollo de software.

URL del Curso: https://n9.cl/gcoronelc-java-fund

Avance del curso: https://n9.cl/gcoronelc-fp-avance

Cupones de descuento: http://gcoronelc.github.io

JAVA ORIENTADO A OBJETOS

CURSO PROFESIONAL DE JAVA ORIENTADO A OBJETOS

Eric Gustavo Coronel Castillo
www.desarrollasoftware.com
INSTRUCTOR

En este curso aprenderás a crear software aplicando la Orientación a Objetos, la programación en capas, el uso de patrones de software y Swing.

Cada tema está desarrollado con ejemplos que demuestran los conceptos teóricos y finalizan con un proyecto aplicativo.

URL del Curso: https://bit.ly/2B3ixUW

Avance del curso: https://bit.ly/2RYGXIt

Cupones de descuento: http://gcoronelc.github.io

PROGRAMACIÓN CON JAVA JDBC

PROGRAMACIÓN DE BASE DE DATOS ORACLE CON JAVA JDBC

Eric Gustavo Coronel Castillo
www.desarrollasoftware.com
INSTRUCTOR

En este curso aprenderás a programas bases de datos Oracle con JDBC utilizando los objetos Statement, PreparedStatement, CallableStatement y a programar transacciones correctamente teniendo en cuenta su rendimiento y concurrencia.

Al final del curso se integra todo lo desarrollado en una aplicación de escritorio.

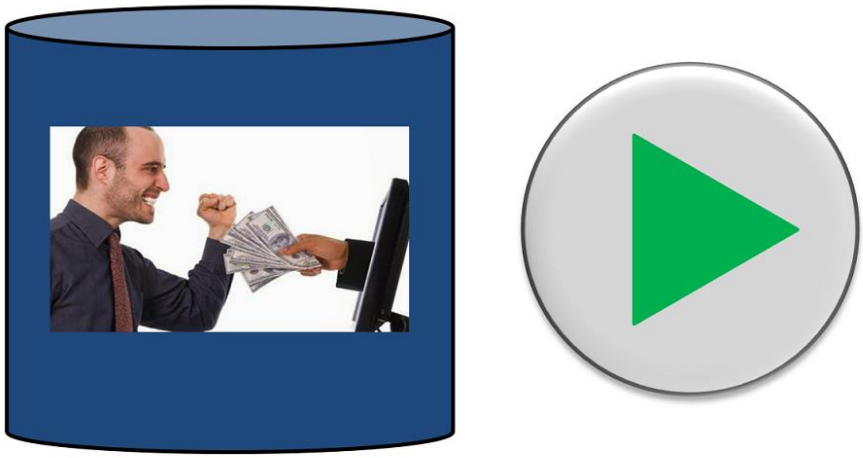
URL del Curso: https://bit.ly/31apy0O

Avance del curso: https://bit.ly/2vatZOT

Cupones de descuento: http://gcoronelc.github.io

PROGRAMACIÓN CON ORACLE PL/SQL

ORACLE PL/SQL



En este curso aprenderás a programas las bases de datos ORACLE con PL/SQL, de esta manera estarás aprovechando las ventas que brinda este motor de base de datos y mejoraras el rendimiento de tus consultas, transacciones y la concurrencia.

Los procedimientos almacenados que desarrolles con PL/SQL se pueden ejecutarlo de Java, C#, PHP y otros lenguajes de programación.

URL del Curso: https://bit.ly/2YZjfxT

Avance del curso: https://bit.ly/3bciqYb

Cupones de descuento: http://gcoronelc.github.io