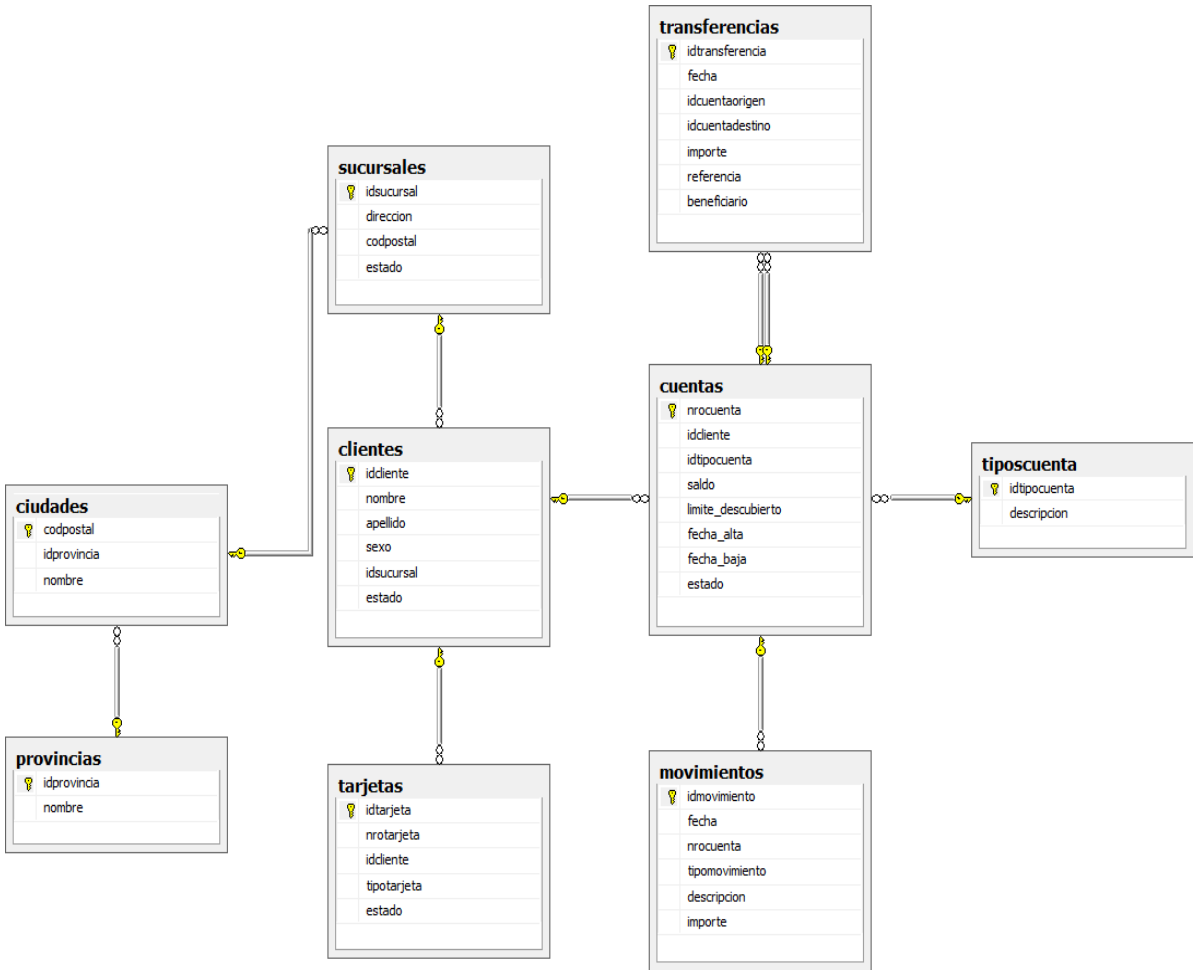
Transacciones

Es muy común que al normalizar nuestra base de datos queramos representar un objeto de la vida real en una tabla. El problema, es que en ocasiones con una sola tabla no es suficiente para representar a una entidad y debemos realizarlo en dos o más tablas.
Sea cual sea el caso, puede ocurrir que una operación en nuestra base de datos requiere modificar más de un registro a la vez.
Aquí ocurre una situación muy crítica en la estructura de la base de datos, obviamente las consultas que se ejecutan en un proceso batch T-SQL deberán ejecutarse una a una. De modo que es posible que nuestra base de datos quede en un estado inconsistente.
Esto se debe a que un conjunto de operaciones que representan una sola operación más grande o general debe ser ejecutada como un todo. Esto se basa en el principio de atomicidad, es decir, debe ser una unidad atómica de trabajo; por lo que se ejecutan todas las operaciones definidas en el proceso o no se ejecuta nada.
Veamos unos ejemplos prácticos donde las consultas deben ser ejecutadas como una transacción.



Cuando se realiza un movimiento sobre la cuenta, por ejemplo cuando se realiza un depósito se debe asentar el movimiento y su posterior actualización del saldo en la cuenta.

Será necesario insertar un registro sobre la tabla de Movimientos indicando la fecha, nro de cuenta, tipo de movimiento, descripción e importe. Este registro permitirá, por ejemplo, elaborar el reporte detallado del estado de cuenta mes a mes.

También será necesario realizar una modificación sobre la tabla Cuentas, actualizando el saldo a su nuevo valor. En el caso del depósito se deberá sumar el importe del mismo al saldo.

La gravedad de ejecutar éstas dos consultas como comandos aislados es que permitiría por ejemplo, registrar un movimiento pero no actualizar el saldo.

Por ejemplo, supongamos que una persona tiene en una cuenta un saldo de $200. Luego realiza un depósito por $10.000 y nuestro procedimiento almacenado que registra esa operación, genera el registro en la tabla de movimientos y en ese momento se corta el suministro eléctrico, sin que se actualice el saldo en la cuenta. Lo que ocurriría es que la base de datos se encontraría en un estado inconsistente y nuestro cliente contaría con $10000 menos en la cuenta.

Esto es un claro ejemplo que ésta operación debe realizarse completamente o no realizarse.

La estructura de una transacción

Las transacciones en SQL Server pueden ser de dos formas, explícitas o implícitas. La forma implícita es la que estuvimos utilizando hasta ahora donde cada consulta individual es confirmada o desecha ni bien finaliza.
La transacción explícita se diferencia porque se especifica en el proceso por lotes el inicio y fin de la misma:

BEGIN TRANSACTION: Indica que comienza el proceso de transacción

COMMIT TRANSACTION: Indica que finaliza la transacción. La misma sentencia intentará hacer permanentes los cambios en la base de datos asentando todas las sentencias que formen parte de la transacción.

ROLLBACK TRANSACTION: Indica que hay que deshacer los cambios sobre los datos ya que ha ocurrido un error y se deberá dejar la base de datos en el estado previo a la ejecución de la transacción.

Veamos el ejemplo que definimos anteriormente, para ello utilizaremos el procedimiento almacenado que figura en el apunte anterior.

```
CREATE PROCEDURE spRegistrarMovimiento(
@nroCuenta VARCHAR(20),
@tipoMovimiento CHAR,
@descripcion VARCHAR(50),
@importe DECIMAL(10, 2)
)
AS
BEGIN
--Comenzamos con el manejo de errores
BEGIN TRY
--Comienza la transacción
BEGIN TRANSACTION
--Verificamos si el tipo de movimiento es Extracción
IF @tipoMovimiento = 'E' OR @tipoMovimiento ='e'
BEGIN
--Declaramos las variables
DECLARE @saldo DECIMAL(10, 2)
DECLARE @tipoCuenta INT
DECLARE @descubierto DECIMAL(10, 2)
--Le asignamos valores que provienen de una consulta SQL
IF (SELECT COUNT(*) FROM cuentas WHERE nroCuenta = @nroCuenta) = 1
BEGIN
```

```
SELECT @saldo = C.saldo, @tipoCuenta = C.idTipoCuenta, @descubierto =
C.limite_descubierto FROM cuentas C WHERE C.nroCuenta = @nroCuenta
END
ELSE
BEGIN
RAISERROR('NO EXISTE LA CUENTA', 16, 1)
END

--Verificamos si se puede hacer la Extracción
IF @saldo - @importe < 0 AND @tipoCuenta = 1
BEGIN
RAISERROR ('NO SE PUEDE REALIZAR EL MOVIMIENTO, SALDO ES INSUFICIENTE', 16, 1)
END
IF @saldo - @importe < (0 - @descubierto) AND @tipoCuenta = 2
BEGIN
RAISERROR ('NO SE PUEDE REALIZAR EL MOVIMIENTO, SALDO ES INSUFICIENTE', 16, 1)
END
END

--Registramos el movimiento en la tabla de Movimientos
INSERT INTO movimientos (fecha, nrocuenta, descripcion, tipomovimiento,
importe) VALUES(GETDATE(), @nroCuenta, @descripcion, @tipoMovimiento,
@importe)

--Si es Extracción el importe debe restarse
IF @tipoMovimiento = 'E' OR @tipoMovimiento = 'e'
BEGIN
SET @importe = @importe * -1
END

--Actualizamos el saldo de la cuenta
UPDATE cuentas SET saldo = saldo + @importe WHERE nroCuenta = @nroCuenta
-- Si no actualizó ninguna fila
IF @@ROWCOUNT = 0
BEGIN
RAISERROR('OCURRIO UN ERROR', 16, 1)
END

COMMIT TRANSACTION
END TRY
BEGIN CATCH
PRINT ERROR_MESSAGE()
ROLLBACK TRANSACTION
END CATCH
--Finaliza el manejo de errores
END
```

Ahora sí, con la consulta anterior podemos estar seguros de que nuestro procedimiento almacenado spRegistrarMovimiento podrá encargarse de la gestión de un depósito o una extracción y tendrá un manejo de errores adecuado en caso de que ocurriera algún imprevisto.

Se observa que la estructura de la transacción es la que nos asegura dicho funcionamiento. Se caracteriza por tener la siguiente forma:

```
BEGIN TRANSACTION
-- instrucción T-SQL 1
-- instrucción T-SQL 2
-- instrucción T-SQL 3
COMMIT TRANSACTION
ó
ROLLBACK TRANSACTION /* Si ocurrió algún error */
```

Dicha estructura tiene más sentido acompañada de un bloque TRY...CATCH que se encargará de la gestión de errores. Hay que tener en cuenta que TRY...CATCH no captura los siguientes errores:
- Errores producidos por error de sintaxis.
- Mensajes de advertencia
- Errores que finalizan el proceso por lotes o el procedimiento por parte del motor de SQL Server.
- Errores por la cancelación de la conexión por parte de la aplicación

La función global @@ROWCOUNT

Cabe destacar que en el caso de la operación de actualización de saldo no se encuentre el nro de cuenta, por alguna razón. Dicha operación no fallará sino que generará un UPDATE que afecta a cero filas. Esto no es considerado un error por lo que no será descartado hacia el CATCH. Es por eso que se hace uso de la función global @@ROWCOUNT, la misma contiene la cantidad de filas afectadas por la última sentencia SQL ejecutada. Es por eso que si dicha variable contiene 0 significará que no se pudo realizar el UPDATE correctamente o que éste no afectó ninguna fila.

La función global @@IDENTITY

Para ver mejor aún el concepto de transacción vamos a ver un ejemplo en el que sea necesario la modificación de más de una tabla en un mismo proceso. Supongamos que deseamos realizar un procedimiento almacenado para crear un usuario y que, como regla de negocio, el banco decida gestionarle automáticamente una cuenta del tipo caja de ahorro y una tarjeta de débito.

Los pasos a seguir de este procedimiento almacenado son:

- Dar de alta el usuario
- Dar de alta la cuenta relacionada al usuario

- Dar de alta la tarjeta de débito relacionada al usuario

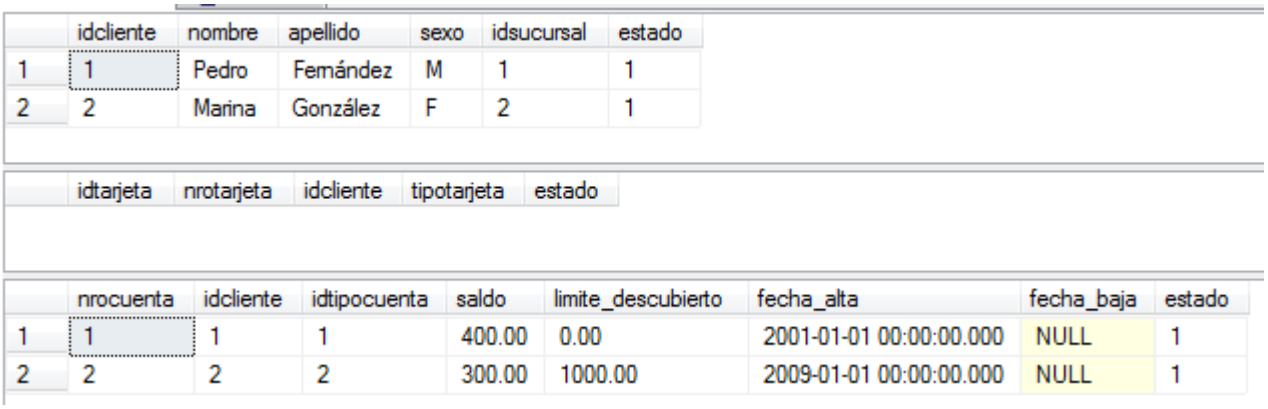
El problema que tenemos aquí, es el siguiente: ya podemos solucionar que el conjunto de procesos SQL a ejecutar se ejecuten como una unidad de procesamiento. Es decir, o todos o ninguno.
Sin embargo, el idcliente en la tabla de clientes es autogenerado y tanto la tabla cuentas como la de tarjetas necesitarán dicho idcliente generado para poder realizar la relación foránea con la tabla de Clientes.
Es aquí donde, dentro de una transacción, que podemos hacer uso de la función global @@identity. Dentro de ésta función global obtendremos el último valor autonumérico generado.

Por lo tanto, nuestro procedimiento almacenado spCrearUsuario quedaría de la siguiente manera:

```
CREATE PROCEDURE spAgregarCliente(
@nombre VARCHAR(50),
@apellido VARCHAR(50),
@sexo CHAR,
@idsucursal INT
)
AS
BEGIN
BEGIN TRANSACTION
BEGIN TRY
-- Generamos el usuario
INSERT INTO Clientes (nombre, apellido, sexo, idsucursal, estado) VALUES
(@nombre, @apellido, @sexo, @idsucursal, 1)
-- Generamos la cuenta del usuario
DECLARE @nroCuenta BIGINT
SELECT @nroCuenta = MAX(nrocuenta) FROM cuentas

DECLARE @idCliente BIGINT
SET @idCliente = @@IDENTITY

INSERT INTO Cuentas (nrocuenta, idcliente, idtipocuenta, saldo,
limite_descubierto, fecha_alta, fecha_baja, estado) VALUES (@nroCuenta+1,
@idCliente, 1, 0, 0, GETDATE(), NULL, 1)

--Generamos la tarjeta de débito
INSERT INTO Tarjetas (nrotarjeta, idcliente, tipotarjeta, estado) VALUES ('D-'
+ CONVERT(NVARCHAR(10), @idCliente) + '-1', @idCliente, 'D', 1)

COMMIT TRANSACTION
END TRY
BEGIN CATCH
PRINT 'QUÉ RARO, EN MI CASA FUNCIONA'
ROLLBACK TRANSACTION
END CATCH
END
```

Veamos el funcionamiento del procedimiento almacenado anterior, antes que nada vamos a ejecutar las siguientes consultas de SELECT para poder conocer el contenido de las tablas clientes, tarjetas y cuentas.

```
SELECT * FROM clientes
SELECT * FROM tarjetas
SELECT * FROM cuentas
```



	idcliente	nombre	apellido	sexo	idsucursal	estado
1	1	Pedro	Fernández	M	1	1
2	2	Marina	González	F	2	1

	idtarjeta	nrotarjeta	idcliente	tipotarjeta	estado

	nrocuenta	idcliente	idtipocuenta	saldo	limite_descubierto	fecha_alta	fecha_baja	estado
1	1	1	1	400.00	0.00	2001-01-01 00:00:00.000	NULL	1
2	2	2	2	300.00	1000.00	2009-01-01 00:00:00.000	NULL	1

Ahora procedemos a ejecutar nuestro Stored Procedure. Para ello ejecutaremos

```
EXEC spAgregarCliente 'Angel', 'Simon', 'M', 1
```

La ejecución de dicho procedimiento almacenado englobará un conjunto de procesos. En principio generará el cliente con código de cliente autonumérico. Luego generará un número de tarjeta con el formato 'Tipo de tarjeta-Id Cliente-Cantidad de tarjetas x cliente'. Donde el tipo de tarjeta, en este caso, siempre será 'D' por débito, el código de cliente surgirá del autogenerado en la consulta anterior y almacenado en la variable @idcliente pero obtenido anteriormente por la función global @@identity y la cantidad de tarjetas por cliente será uno ya que se está generando el cliente en el momento. Y por último, generará una cuenta del tipo caja de ahorro con saldo cero y límite de descubierto cero para dicho cliente. El número de cuenta se calcula en base al número de cuenta máximo generado anteriormente más uno. Nuevamente el idcliente surge de nuestra variable @idcliente y previamente por @@identity y los demás valores son asignados de manera constante salvo por la fecha que se obtiene con GETDATE().
Luego de ejecutar el procedimiento almacenado si obtenemos los datos de nuestras tres tablas obtendremos:

	idcliente	nombre	apellido	sexo	idsucursal	estado
1	1	Pedro	Femández	M	1	1
2	2	Marina	González	F	2	1
3	5	Angel	Simon	M	1	1

	idtarjeta	nrotarjeta	idcliente	tipotarjeta	estado
1	3	D-5-1	5	D	1

	nrocuenta	idcliente	idtipocuenta	saldo	limite_descubierto	fecha_alta	fecha_baja	estado
1	1	1	1	400.00	0.00	2001-01-01 00:00:00.000	NULL	1
2	2	2	2	300.00	1000.00	2009-01-01 00:00:00.000	NULL	1
3	3	5	1	0.00	0.00	2012-05-29 22:37:45.613	NULL	1