

Unidad 5

Concurrencia

- 5.1 Entregas Automáticas y Transacciones Explícitas.
- 5.2 COMMIT y ROLLBACK.
- 5.3 Propiedades de las transacciones.
- 5.4 Niveles de aislamiento.
- 5.5 Interbloqueo (*DeadLock*).

5.1 Entregas Automáticas vs Transacciones Explícitas

Cada vez que se ejecuta INSERT, UPDATE, DELETE, los datos se escriben a las tablas de la Base de Datos inmediatamente.

Nosotros diremos que se ***entregan*** (*commit*).

Con frecuencia, esta palabra en inglés se traduce en libros de texto como *comprometer*; *confirmar* es una palabra más adecuada, sin embargo, en este curso la traduciremos como ***entregar***.

5.1 Entregas Automáticas vs Transacciones Explícitas.

Este proceso de entrega automática, es conveniente en algunos casos y en otros no.

Para estudiar los fenómenos causados por la concurrencia, empezaremos analizando algunas situaciones para comprobar que el ***automatic commit*** no siempre produce buenos resultados.

5.1 Entregas Automáticas vs Transacciones Explícitas.

Ejemplo 1

1. Trabajaremos con el siguiente procedimiento almacenado que ya tenemos creado:

sp_TraspasoPresupuesto

2. Recordemos que el presupuesto total asignado para el ITD fue de 3,350,000.00 para el año 2024. Lo podemos consultar mediante la función siguiente:

SELECT dbo.fn_PresupuestoTotal(2024)

5.1 Entregas Automáticas vs Transacciones Explícitas.

3. El presupuesto detallado para cada una de las áreas administrativas se puede consultar de la siguiente forma:

```
SELECT * FROM vPresupuestoAnual
```

(Hay que crear la vista con los atributos *idAreaAdmin*, *Nombre*, *Tipo* y *Monto* de las tablas *AreasAdministrativas* y *PresupuestoAnual*).

4. Para hacer los ejercicios, deberemos hacer **una pequeña modificación** al procedimiento almacenado, en la diapositiva siguiente se muestra el cambio que hay que efectuar.

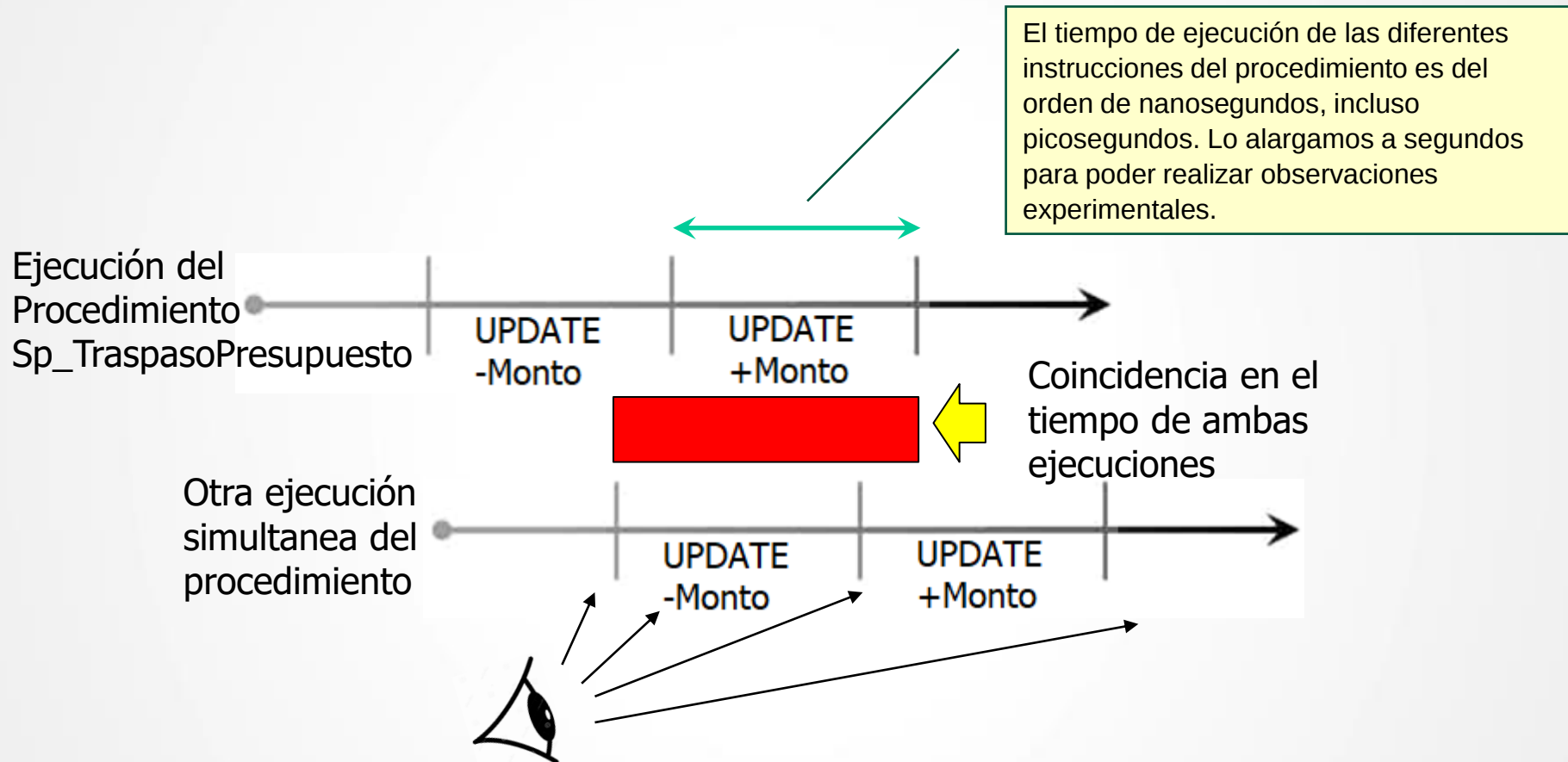
5.1 Entregas Automáticas vs Transacciones Explícitas.

Ejemplo 1

```
ALTER PROCEDURE sp_TraspasoPresupuesto
    @vIdAreaOrigen INT,
    @vIdAreaDestino INT,
    @vAnio INT,
    @vImporteMover NUMERIC(12,2) AS
BEGIN
    BEGIN TRY
        UPDATE PresupuestoAnual
            SET Monto=Monto-@vImporteMover
            WHERE idAreaAdmin=@vIdAreaOrigen and Anio=@vAnio
        WAITFOR DELAY '00:01:00'
        UPDATE PresupuestoAnual
            SET Monto=Monto+@vImporteMover
            WHERE idAreaAdmin=@vIdAreaDestino and Anio=@vAnio
    END TRY
    BEGIN CATCH
        IF ERROR_MESSAGE() like '%CK_PresupuestoMonto%'
            THROW 50001, '¡Monto insuficiente para el traspaso!', 1
    END CATCH
END
```

“Alargamos” el tiempo para observar lo que puede pasar en cualquier momento cuando hay acceso simultaneo.

5.1 Entregas Automáticas vs Transacciones Explícitas.



Por otro lado, un “observador” (una consulta) puede ver en cualquier instante los cambios a los datos cuando se realizan *automatic commits* (entregas automáticas).

5.1 Entregas Automáticas vs Transacciones Explícitas.

Si consideramos que el presupuesto anual asignado para el ITD para el año 2024 es de 3,350,000 y no es posible modificar el importe total, se dice que la Base de Datos se encuentra en un **estado consistente** y debe mantenerse de esa forma ya que como lo vimos antes, se pueden hacer traspasos entre las diferentes áreas pero sin modificar el monto total asignado.

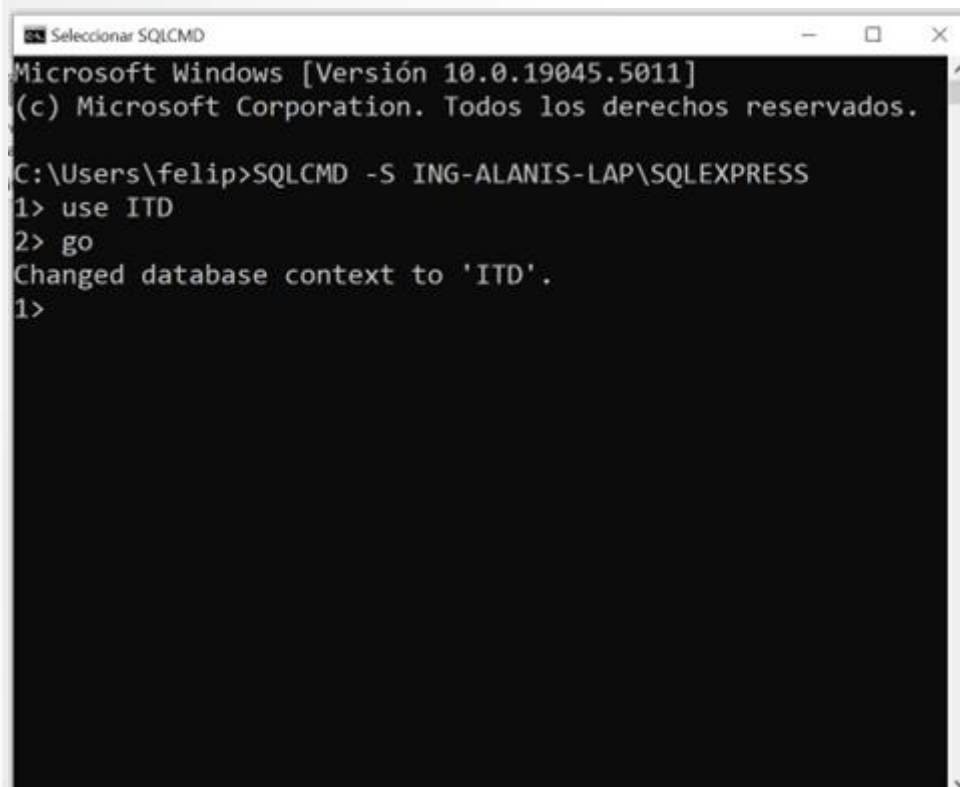
Trabajaremos con acceso simultaneo para observar el comportamiento de los procedimientos, por lo que requerimos acceder a los datos concurrentemente.

5.1 Entregas Automáticas vs Transacciones Explícitas.

Abra dos sesiones, una de consola (*CMD*) y otra de SSMS.
Para la sesión de consola acceda de la siguiente forma:

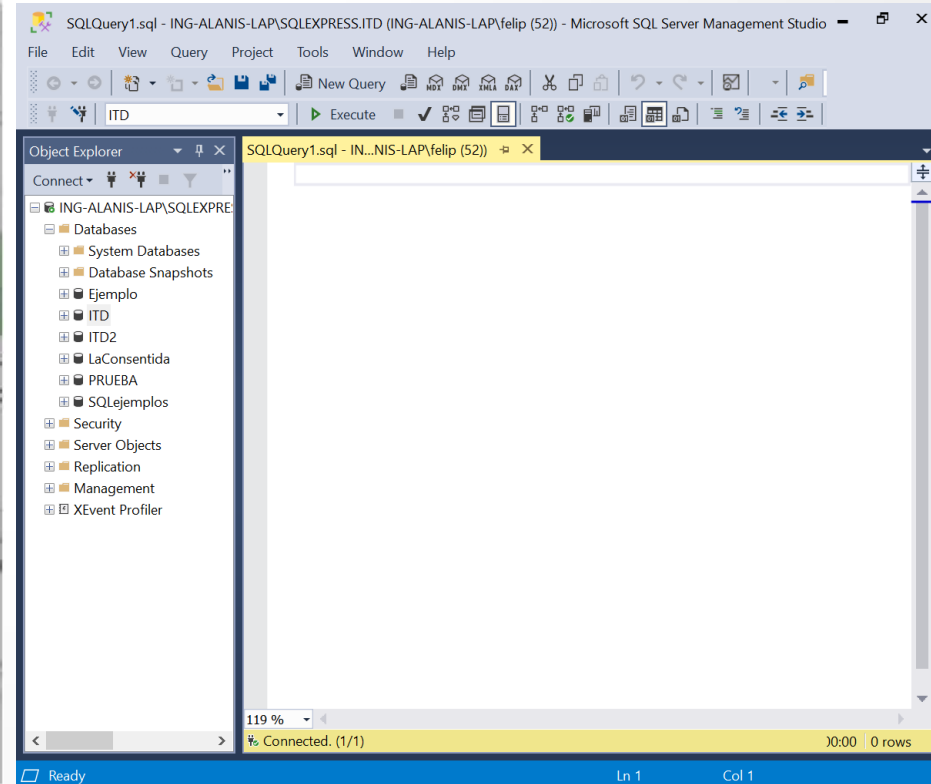
SQLCMD -S <NOMBRE DEL SERVIDOR DE BD>

P.EJ.: SQLCMD -S ING-ALANIS-LAP\SQLEXPRESS



```
Microsoft Windows [Versión 10.0.19045.5011]
(c) Microsoft Corporation. Todos los derechos reservados.

C:\Users\felip>SQLCMD -S ING-ALANIS-LAP\SQLEXPRESS
1> use ITD
2> go
Changed database context to 'ITD'.
1>
```



5.1 Entregas Automáticas vs Transacciones Explícitas.

Ejecute los comandos en la sesión que se indica.

Consola:

```
EXECUTE dbo.sp_TraspasoPresupuesto 1,2,2024,10000
```

SSMS:

```
SELECT dbo.fn_PresupuestoTotal(2024)
```

Espera más de un minuto y vuelve a ejecutar:

SSMS:

```
SELECT dbo.fn_PresupuestoTotal(2024)
```

5.1 Entregas Automáticas vs Transacciones Explícitas.

La primera consulta hecha en el **SSMS** informa que el presupuesto total es diferente al presupuesto correcto.

La causa es que los datos son entregados inmediatamente (*Automatic Commit*) y la consulta se hace luego de que se ejecuta la disminución al presupuesto desde el área origen, pero antes del aumento al área destino (lo podemos ver gracias al retraso que añadimos al procedimiento almacenado).

5.1 Entregas Automáticas vs Transacciones Explícitas.

Al hacer la segunda consulta, el presupuesto se mostrará en forma correcta.

El hecho de que durante cierto tiempo la BD se encuentre en un estado inconsistente, representa un inconveniente ya que esa consulta errónea puede ocasionar la toma de una o más **decisiones incorrectas**.

5.1 Entregas Automáticas vs Transacciones Explícitas.

Sin el retraso que añadimos al procedimiento almacenado, la inconsistencia se puede presentar igualmente si la consulta en la segunda sesión se hace justo fracciones de segundo después del primer ***update***, pero antes del segundo.

5.1 Entregas Automáticas vs Transacciones Explícitas.

Ejemplo 2a

Ahora ejecute la siguiente secuencia para comprobar que la inconsistencia ya no sería temporal, sino definitiva en caso de una falla, lo que constituye un riesgo grave.

Consola o SSMS:

```
EXECUTE dbo.sp_TraspasoPresupuesto 2,3,2024,20000
```

Si trabaja en una PC, desconecte el cable de poder simulando una falla (en una Laptop detenga SQL Server). Luego encienda el equipo o inicie de nuevo SQL Server y consulte el presupuesto total.

Puede ser una falla de hardware, de energía o una excepción controlada o no.

Consola o SSMS:

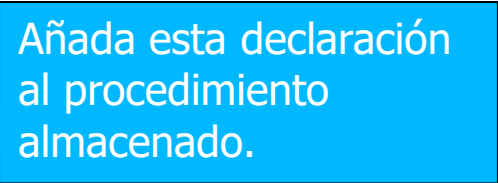
Usando SQL Server Configuration Manager

```
SELECT dbo.fn_PresupuestoTotal(2024)
SELECT * FROM vPresupuestoAnual
```

5.1 Entregas Automáticas vs Transacciones Explícitas.

Ejemplo 2b

```
ALTER PROCEDURE sp_TraspasoPresupuesto
    @vIdAreaOrigen INT,
    @vIdAreaDestino INT,
    @vAnio INT,
    @vImporteMover NUMERIC(12,2) AS
BEGIN
    BEGIN TRY
        BEGIN TRANSACTION
        UPDATE PresupuestoAnual
            SET Monto=Monto-@vImporteMover
            WHERE idAreaAdmin=@vIdAreaOrigen and Anio=@vAnio
        WAITFOR DELAY '00:01:00'
        UPDATE PresupuestoAnual
            SET Monto=Monto+@vImporteMover
            WHERE idAreaAdmin=@vIdAreaDestino and Anio=@vAnio
        END TRY
        BEGIN CATCH
            IF ERROR_MESSAGE() like '%CK_PresupuestoMonto%'
                THROW 50001, '¡Monto insuficiente para el traspaso!', 1
        END CATCH
    END
```



5.1 Entregas Automáticas vs Transacciones Explícitas.

Corrija el monto del presupuesto que quedó incompleto para que el total sea de nuevo 3,350,000.00, luego ejecute la siguiente secuencia (similar al ejercicio anterior) y observe el comportamiento.

Consola o SSMS:

```
EXECUTE dbo.sp_TraspasoPresupuesto 3,4,2024,30000
```

Si trabaja en una PC desconecte el cable de poder (en una Laptop detenga SQL Server). Luego encienda el equipo o inicie SQL Server y consulte el presupuesto total.

Usando SQL Server
Configuration Manager

Consola o SSMS:

```
SELECT dbo.fn_PresupuestoTotal(2024)  
SELECT * FROM vPresupuestoAnual
```

5.2 COMMIT y ROLLBACK

Ejemplo 3

(Haga estas pruebas solo en una sesión)

```
ALTER PROCEDURE sp_TraspasoPresupuesto
```

```
    @vIdAreaOrigen INT,
```

```
    @vIdAreaDestino INT,
```

```
    @vAnio INT,
```

```
    @vImporteMover NUMERIC(12,2) AS
```

```
BEGIN
```

```
    BEGIN TRY
```

```
        BEGIN TRANSACTION
```

```
        UPDATE PresupuestoAnual
```

```
        SET Monto=Monto-@vImporteMover
```

```
        WHERE idAreaAdmin=@vIdAreaOrigen and Anio=@vAnio
```

```
        UPDATE PresupuestoAnual
```

```
        SET Monto=Monto+@vImporteMover
```

```
        WHERE idAreaAdmin=@vIdAreaDestino and Anio=@vAnio
```

```
        COMMIT
```

```
        WAITFOR DELAY '00:01:00'
```

```
    END TRY
```

```
    BEGIN CATCH
```

```
        ROLLBACK
```

```
        IF ERROR_MESSAGE() like '%CK_PresupuestoMonto%'
```

```
            THROW 50001, '¡Monto insuficiente para el traspaso!', 1
```

```
    END CATCH
```

```
END
```

REALICE LAS PRUEBAS INDICADAS
USANDO LOS EJEMPLOS DE LA
PAGINA SIGUIENTE

EN ESTE PUNTO DETENGA
SQL SEVER. LUEGO
REINICIELO Y CONSULTE
EL ESTADO DEL
PRESUPUESTO

Retire los *WAITFOR DELAY* y haga las pruebas de la página siguiente con y sin *ROLLBACK*. *ROLLBACK* termina la transacción sin entregar los datos, *COMMIT* termina la transacción y entrega los datos. Una de las pruebas intenta traspasar un monto excesivo.

5.2 COMMIT y ROLLBACK

Use los ejemplos siguientes para que pruebe los casos que se indican en la diapositiva anterior.

Consola o SSMS:

```
EXECUTE dbo.sp_TraspasoPresupuesto 3,4,2024,30000  
SELECT dbo.fn_PresupuestoTotal(2024)  
SELECT * FROM vPresupuestoAnual
```

Consola o SSMS:

```
EXECUTE dbo.sp_TraspasoPresupuesto 1,2,2024,1000000  
SELECT dbo.fn_PresupuestoTotal(2024)  
SELECT * FROM vPresupuestoAnual
```

5.2 COMMIT y ROLLBACK

Ejemplo 4

Ejecute la creación del procedimiento de la diapositiva siguiente, una vez creado ejecútelo bajo las condiciones que tambien se indican.

Es una transacción que se compone de dos INSERT correspondiente a un ejercicio planteado en una Unidad anterior. Las transacciones pueden contener cualquier cantidad y combinación de operaciones INSERT, UPDATE, DELETE.

```

CREATE PROCEDURE sp_AltaAlumno
    @vNombre VARCHAR(MAX), @vApellidos VARCHAR(MAX),
    @vCalle VARCHAR(MAX), @vNumExt INT, @vPoblacion VARCHAR(MAX),
    @vPais VARCHAR(MAX), @vfechaNac DATE, @vSexo VARCHAR(MAX),
    @vCurp VARCHAR(MAX), @vNumControl VARCHAR(MAX),
    @vEscuelaProcede VARCHAR(MAX) AS
BEGIN
    BEGIN TRY
        BEGIN TRANSACTION
        INSERT INTO Personas
            (Nombre, Apellidos, Calle, NumExt, Poblacion, Pais,
            FechaNac, Sexo, Curp)
        VALUES
            (@vNombre, @vApellidos, @vCalle, @vNumExt, @vPoblacion, @vPais,
            @vfechaNac, @vSexo, @vCurp)
        DECLARE @vIdPersona INT
        SET @vIdPersona = SCOPE_IDENTITY()
        INSERT INTO Alumnos (NumControl, EscuelaProcede, idPersona)
        VALUES (@vNumControl, @vEscuelaProcede, @vIdPersona)
        COMMIT
    END TRY
    BEGIN CATCH
        ROLLBACK
        IF ERROR_MESSAGE() LIKE '%UQ_Personas_Curp%'
            THROW 50001, '¡Ya existe una Persona con esa CURP!', 1
        ELSE IF ERROR_MESSAGE() LIKE '%UQ_Alumnos_NumControl%'
            THROW 50001, '¡Ya existe un Alumno con ese Número de Control!', 1
    END CATCH
END

```

Devuelve el último valor de identidad insertado en una columna de identidad en el mismo ámbito (este procedimiento almacenado).

```
SELECT * FROM Alumnos
SELECT * FROM Personas
```

```
EXECUTE sp_AltaAlumno 'José', 'González',
'Cerro de la Bufo', 106, 'Durango', 'México',
'1961-01-01', 'Hombre', 'JG0101', '76040150',
'ITD'
```

```
SELECT * FROM Alumnos
SELECT * FROM Personas
```

```
EXECUTE sp_AltaAlumno 'Guadalupe', 'Pineda',
'Pasteur', 222, 'Durango', 'México',
'1963-05-02', 'Mujer', 'JG0101', '76040999',
'ITD'
```

```
EXECUTE sp_AltaAlumno 'Josefina', 'Huerta',
'Patoni', 106, 'Durango', 'México',
'1962-02-02', 'Mujer', 'JH0202', '76040150',
'ITD'
```

```
SELECT * FROM Alumnos
SELECT * FROM Personas
```

Llamadas a ejecución.
Observe los resultados.

5.3 Propiedades de las transacciones

Propiedades de las transacciones:

- ☐ **A**tomicidad
- ☐ **C**onsistencia
- ☐ Aislamiento (**I**solation en inglés)
- ☐ **D**urabilidad

Conocidas como propiedades **ACID**.

5.3 Propiedades de las transacciones

1. Atomicidad.

Se requiere que **todas** las operaciones de una transacción sean entregadas (*committed*) a la base de datos.

En caso de que una de ellas no sea entregada por cualquier razón, ninguna debe hacerse.

5.3 Propiedades de las transacciones

2. Consistencia.

Se requiere que, ya que la BD estará consistente antes de ejecutar una transacción, **permanezca** igualmente consistente una vez terminada la transacción.

5.3 Propiedades de las transacciones

3. Aislamiento.

Si se presenta la ejecución concurrente de transacciones, cada una de ellas debe tener la impresión de que las demás **no existen**, como si se ejecutaran en serie (una después de la otra).

5.3 Propiedades de las transacciones

4. Durabilidad.

Es indispensable que **una vez finalizada** la transacción y los **datos entregados** a la BD, estos no se pierdan incluso aunque se produzca una falla.

5.3 Propiedades de las transacciones

Con la excepción de aislamiento, en los ejercicios anteriores hemos comprobado esas propiedades.

5.4 Niveles de Aislamiento

Los **Niveles de Aislamiento** describen el grado en el cual los datos que están siendo manipulados por una sesión son accesibles a otras, los niveles de aislamiento básicos son:

- ***Serializable*** (En serie).
- ***Repeatable Read*** (Lecturas Repetibles)
- ***Read Committed*** (Lectura de datos entregados).
- ***Read Uncommitted*** (Lectura de datos no entregados).

5.4 Niveles de Aislamiento

Ejemplo 5

```
ALTER PROCEDURE sp_TraspasoPresupuesto
    @vIdAreaOrigen INT,
    @vIdAreaDestino INT,
    @vAnio INT,
    @vImporteMover NUMERIC(12,2) AS
BEGIN
    BEGIN TRY
        BEGIN TRANSACTION
        WAITFOR DELAY '00:00:30'
        UPDATE PresupuestoAnual
        SET Monto=Monto-@vImporteMover
        WHERE idAreaAdmin=@vIdAreaOrigen and Anio=@vAnio
        WAITFOR DELAY '00:00:30'
        UPDATE PresupuestoAnual
        SET Monto=Monto+@vImporteMover
        WHERE idAreaAdmin=@vIdAreaDestino and Anio=@vAnio
        COMMIT
    END TRY
    BEGIN CATCH
        ROLLBACK
        IF ERROR_MESSAGE() like '%CK_PresupuestoMonto%'
            THROW 50001, '¡Monto insuficiente para el traspaso!', 1
    END CATCH
END
```

Añada este WAITFOR
DELAY

Hay que conservar
este WAITFOR DELAY

5.4 Niveles de Aislamiento

DBCC USEROPTIONS

Consola:

La opción *isolation level* debe contener el valor READ COMMITTED, en caso de que no sea así, ejecute el siguiente comando:
SET TRANSACTION ISOLATION LEVEL READ COMMITTED

SSMS:

EXECUTE dbo.sp_TraspasoPresupuesto 3,4,2024,30000

Consola:

```
SELECT dbo.fn_PresupuestoTotal(2024)
SELECT * FROM vPresupuestoAnual
GO
```

Observe que estas dos expresiones de consulta dan los resultados de inmediato

-- ESPERE 35 SEGUNDOS ANTES DE EJECUTAR

-- LAS MISMAS CONSULTAS DE NUEVO

```
SELECT dbo.fn_PresupuestoTotal(2024)
SELECT * FROM vPresupuestoAnual
GO
```

Por el contrario, al ejecutar de nuevo las mismas consultas, en esta ocasión tarda unos segundos en dar los resultados

5.4 Niveles de Aislamiento

Bloqueos Exclusivos (X-LOCK).

- UPDATE: Cuando se actualiza una tupla, se emite un bloqueo exclusivo para evitar que otras transacciones la lean o modifiquen.
- DELETE: Al eliminar una tupla, se emite un bloqueo exclusivo para asegurar que esa tupla no sea leída ni modificada por otras transacciones durante el proceso de eliminación.
- INSERT: Al insertar una nueva tupla, se emite un bloqueo exclusivo sobre ella.
- Solo un proceso a la vez puede emitir un bloqueo exclusivo para una tupla o grupo de tuplas.

5.4 Niveles de Aislamiento

- En un bloque BEGIN TRANSACTION, el X-LOCK persiste hasta el final de la transacción (COMMIT o ROLLBACK).
 - Si no hay un bloque BEGIN TRANSACTION (es decir con entregas automáticas), el bloqueo se emite pero se libera de inmediato en cuanto termina la operación.
- En los niveles de aislamiento REPEATABLE READ, READ COMMITTED y SERIALIZABLE, las tuplas con un X-LOCK no pueden ser consultadas, menos modificadas por otra hasta en tanto la primera termine.

EN EL NIVEL **READ UNCOMMITTED** NO SE EMITEN BLOQUEOS EXCLUSIVOS

5.4 Niveles de Aislamiento

Ejemplo 6a

```
ALTER PROCEDURE sp_TraspasoPresupuesto
    @vIdAreaOrigen INT,
    @vIdAreaDestino INT,
    @vAnio INT,
    @vImporteMover NUMERIC(12,2) AS
BEGIN
    BEGIN TRY
        BEGIN TRANSACTION
        WAITFOR DELAY '00:00:30'
        UPDATE PresupuestoAnual
        SET Monto=Monto-@vImporteMover
        WHERE idAreaAdmin=@vIdAreaOrigen and Anio=@vAnio
        WAITFOR DELAY '00:00:30'
        UPDATE PresupuestoAnual
        SET Monto=Monto+@vImporteMover
        WHERE idAreaAdmin=@vIdAreaDestino and Anio=@vAnio
        COMMIT
    END TRY
    BEGIN CATCH
        ROLLBACK
        IF ERROR_MESSAGE() like '%CK_PresupuestoMonto%'
            THROW 50001, '¡Monto insuficiente para el traspaso!', 1
    END CATCH
END
```

Conserve este
WAITFOR DELAY

Conserve también este
WAITFOR DELAY

5.4 Niveles de Aislamiento

Consola:

```
SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED  
DBCC USEROPTIONS
```

Compruebe el nivel de aislamiento
READ UNCOMMITTED

SSMS:

```
EXECUTE dbo.sp_TraspasoPresupuesto 3,4,2024,30000
```

Consola:

```
SELECT dbo.fn_PresupuestoTotal(2024)  
SELECT * FROM vPresupuestoAnual  
GO  
-- ESPERE 35 SEGUNDOS ANTES DE LAS  
-- SIGUIENTES CONSULTAS  
SELECT dbo.fn_PresupuestoTotal(2024)  
SELECT * FROM vPresupuestoAnual  
GO
```

Observe que primero
se ve consistente las
BD, luego se pierde la
consistencia

Cambio para Ejemplos 6b y 6c

```
ALTER PROCEDURE sp_TraspasoPresupuesto
    @vIdAreaOrigen INT,
    @vIdAreaDestino INT,
    @vAnio INT,
    @vImporteMover NUMERIC(12,2) AS
BEGIN
    BEGIN TRY
        BEGIN TRANSACTION
        WAITFOR DELAY '00:00:30'
        UPDATE PresupuestoAnual
        SET Monto=Monto-@vImporteMover
        WHERE idAreaAdmin=@vIdAreaOrigen and Anio=@vAnio
        WAITFOR DELAY '00:00:30'
        IF (SELECT COUNT(*) FROM PresupuestoAnual WHERE Monto=0)>1
            THROW 50000, '¡No pueden quedar 2 presupuestos en CERO!',1
        UPDATE PresupuestoAnual
        SET Monto=Monto+@vImporteMover
        WHERE idAreaAdmin=@vIdAreaDestino and Anio=@vAnio
        COMMIT
    END TRY
    BEGIN CATCH
        ROLLBACK
        IF ERROR_MESSAGE() like '%CK_PresupuestoMonto%'
            THROW 50001, '¡Monto insuficiente para el traspaso!',1
        ELSE
            BEGIN
                DECLARE @MENSAJE VARCHAR(100)
                SET @MENSAJE = ERROR_MESSAGE()
                THROW 50001, @MENSAJE, 1
            END
    END CATCH
END
```

Modifique el procedimiento almacenado, se añade una restricción que impide que dos áreas queden con presupuesto=0

Ejemplo 6b

Consola:

```
SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED  
DBCC USEROPTIONS
```

SSMS:

```
EXECUTE dbo.sp_TraspasoPresupuesto 2,4,2024,50000
```

Consola:

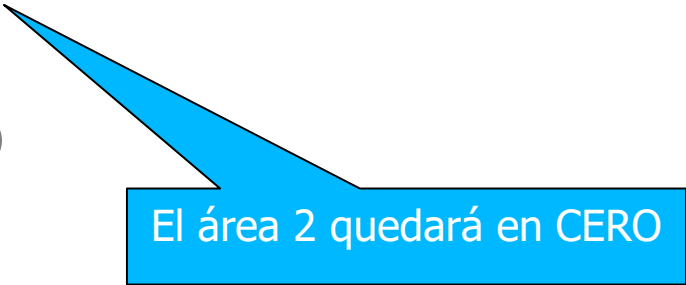
```
SELECT dbo.fn_PresupuestoTotal(2024)  
SELECT * FROM vPresupuestoAnual  
GO
```

```
-- ESPERE 35 SEGUNDOS ANTES DE LAS  
-- SIGUIENTES CONSULTAS
```

```
SELECT dbo.fn_PresupuestoTotal(2024)  
SELECT * FROM vPresupuestoAnual  
GO
```

```
-- ESPERE OTROS 35 SEGUNDOS
```

```
SELECT dbo.fn_PresupuestoTotal(2024)  
SELECT * FROM vPresupuestoAnual  
GO
```



El área 2 quedará en CERO

Ejemplo 6c

Consola:

```
SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED  
DBCC USEROPTIONS
```

SSMS:

```
EXECUTE dbo.sp_TraspasoPresupuesto 3,4,2024,50000
```

Consola:

```
SELECT dbo.fn_PresupuestoTotal(2024)  
SELECT * FROM vPresupuestoAnual  
GO
```

```
-- ESPERE 35 SEGUNDOS ANTES DE LAS  
-- SIGUIENTES CONSULTAS
```

```
SELECT dbo.fn_PresupuestoTotal(2024)  
SELECT * FROM vPresupuestoAnual  
GO
```

```
-- ESPERE OTROS 35 SEGUNDOS
```

```
SELECT dbo.fn_PresupuestoTotal(2024)  
SELECT * FROM vPresupuestoAnual  
GO
```

En el ejemplo 6b el área 2 queda en CERO, ahora se intenta quedar en CERO el área 3

¿Se observan cambios que luego se revierten?

5.4 Niveles de Aislamiento

En el nivel de aislamiento **READ UNCOMMITTED**, se pueden hacer consultas sobre las tuplas que están siendo modificadas porque **no se emite un X-LOCK**.

- Debido a lo anterior, si la transacción termina con ROLLBACK, a la **lectura** que se había obtenido y que **no persistirá** se le llama **LECTURA SUCIA**.
- Bajo este nivel de aislamiento, al no emitirse un X-LOCK, cualquier proceso puede modificar cualquier tupla, con los riesgos de inconsistencia que conlleva.
- Aunque este nivel de aislamiento puede resultar en consultas imprecisas, puede ser conveniente usarlo cuando se requieren resultados estadísticos que no requieren exactitud. Se pueden obtener resultados globales que son muy cercanos a la realidad sin tener que esperar a que terminen las transacciones.

5.4 Niveles de Aislamiento

Ejemplo 7

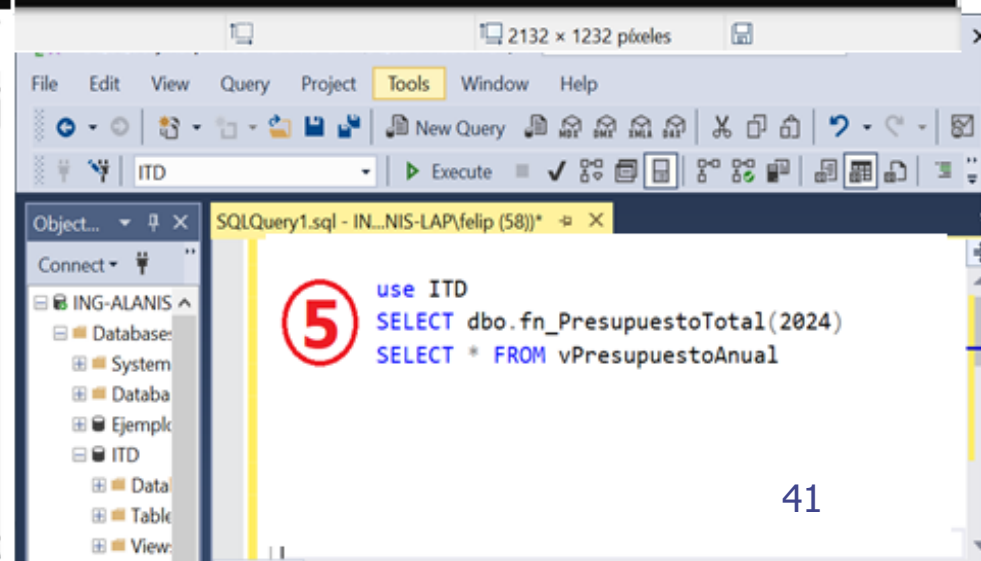
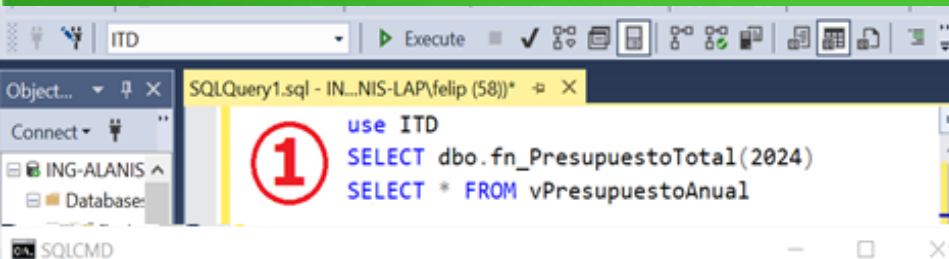
```
ALTER PROCEDURE sp_TraspasoPresupuesto
    @vIdAreaOrigen INT,
    @vIdAreaDestino INT,
    @vAnio INT,
    @vImporteMover NUMERIC(12,2) AS
BEGIN
    BEGIN TRY
        BEGIN TRANSACTION
        UPDATE PresupuestoAnual
        SET Monto=Monto-@vImporteMover
        WHERE idAreaAdmin=@vIdAreaOrigen and Anio=@vAnio
        WAITFOR DELAY '00:00:20'
        UPDATE PresupuestoAnual
        SET Monto=Monto+@vImporteMover
        WHERE idAreaAdmin=@vIdAreaDestino and Anio=@vAnio
        WAITFOR DELAY '00:00:20'
        COMMIT
    END TRY
    BEGIN CATCH
        ROLLBACK
        IF ERROR_MESSAGE() like '%CK_PresupuestoMonto%'
            THROW 50001, '¡Monto insuficiente para el traspaso!', 1
    END CATCH
END
```

Observe que ambos
WAITFOR DELAY están justo
después de cada uno de los
UPDATE y con un retraso de
20 segundos

- Prepare una sesión de SSMS y tres sesiones de CMD como se muestra en la diapositiva siguiente para simular la ejecución de tres procedimientos en forma concurrente y observar si se mantiene la consistencia.
- Establezca, en todas las sesiones, cualquier nivel de aislamiento excepto READ UNCOMMITTED.
- Ejecute en el orden indicado los siguientes comandos sin esperar mucho tiempo entre una y otra ejecuciones.

1	<code>SELECT</code> dbo.fn_PresupuestoTotal(2024) <code>SELECT</code> * <code>FROM</code> vPresupuestoAnual
2	<code>USE</code> ITD <code>EXECUTE</code> dbo.sp_TraspasoPresupuesto 1,2,2024,10000 <code>GO</code>
3	<code>USE</code> ITD <code>EXECUTE</code> dbo.sp_TraspasoPresupuesto 1,2,2024,20000 <code>GO</code>
4	<code>USE</code> ITD <code>EXECUTE</code> dbo.sp_TraspasoPresupuesto 1,2,2024,30000 <code>GO</code>
5	<code>SELECT</code> dbo.fn_PresupuestoTotal(2024) <code>SELECT</code> * <code>FROM</code> vPresupuestoAnual

5.4 Niveles de Aislamiento



Observe que la **sesión 3** entra en espera porque no puede obtener un bloqueo sobre la misma tupla que la **sesión 2** bloqueó.

El bloqueo de la **sesión 2** permanecerá hasta que termine la transacción y en ese momento la **sesión 3** continuará.

Lo mismo pasa entre las **sesiones 3** y **4**.

En nuestro procedimiento de traspaso, el riesgo de inconsistencia por *LOST UPDATES* es nulo porque no hacemos un *update* basado en valores consultados previamente, en su lugar, actualizamos las tuplas directamente en cada *update*:

SET Atributo=Atributo-Valor.

SET Atributo=Atributo+Valor.

Enseguida veremos un ejemplo de *LOST UPDATES*.

Ejemplo 8

LOST UPDATES (Actualizaciones Perdidas)

```
CREATE PROCEDURE [dbo].[sp_TraspasoPresupuestoV2]
    @vIdAreaOrigen INT,
    @vIdAreaDestino INT,
    @vAnio INT,
    @vImporteMover NUMERIC(12,2) AS
BEGIN
    BEGIN TRY
        BEGIN TRANSACTION
        DECLARE @NuevoMontoOrigen NUMERIC(12,2)
        SELECT @NuevoMontoOrigen=Monto-@vImporteMover FROM PresupuestoAnual
            WHERE idAreaAdmin=@vIdAreaOrigen and Anio=@vAnio
        WAITFOR DELAY '00:00:20'
        UPDATE PresupuestoAnual
            SET Monto=@NuevoMontoOrigen
            WHERE idAreaAdmin=@vIdAreaOrigen and Anio=@vAnio
        DECLARE @NuevoMontoDestino NUMERIC(12,2)
        SELECT @NuevoMontoDestino=Monto+@vImporteMover FROM PresupuestoAnual
            WHERE idAreaAdmin=@vIdAreaDestino and Anio=@vAnio
        WAITFOR DELAY '00:00:20'
        UPDATE PresupuestoAnual
            SET Monto=@NuevoMontoDestino
            WHERE idAreaAdmin=@vIdAreaDestino and Anio=@vAnio
        COMMIT
    END TRY
    BEGIN CATCH
        ROLLBACK
        IF ERROR_MESSAGE() like '%CK_PresupuestoMonto%'
            THROW 50001,'¡Monto insuficiente para el traspaso!',1
    END CATCH
END
```

Añada/modifique las instrucciones marcadas. Se lee el valor actual del presupuesto, se le resta o se le suma según el caso y se guarda en una variable. Luego esa variable se usa en el *update*.

5.4 Niveles de Aislamiento

- EN PRIMER LUGAR COMPRUEBE QUE EL NUEVO
- PROCEDIMIENTO ALMACENADO SE EJECUTA
- CORRECTAMENTE POR UN SOLO USUARIO

Consola/SSMS:

```
SELECT dbo.fn_PresupuestoTotal(2024)
SELECT * FROM vPresupuestoAnual
GO
```

```
EXECUTE dbo.sp_TraspasoPresupuestoV2 1,3,2024,20000
GO
```

```
SELECT dbo.fn_PresupuestoTotal(2024)
SELECT * FROM vPresupuestoAnual
GO
```

5.4 Niveles de Aislamiento

-- AHORA SE SIMULAN TRASPASOS POR DOS USUARIOS

Consola:

```
EXECUTE dbo.sp_TraspasoPresupuestoV2 1,2,2024,10000
```

SSMS:

```
EXECUTE dbo.sp_TraspasoPresupuestoV2 1,3,2024,20000
```

Consola:

-- ESPERE A QUE TERMINEN LAS TRANSACCIONES

```
SELECT dbo.fn_PresupuestoTotal(2024)
```

```
SELECT * FROM vPresupuestoAnual
```

```
GO
```

Una actualización perdida es una anomalía de concurrencia que se produce cuando se actualizan datos a partir de una consulta previa sin considerar que otra transacción pudo haber hecho cambios en los mismos datos.

SQL Server no puede hacer todo por nosotros, nos proporciona las herramientas para controlar la concurrencia pero si escribimos código de manera descuidada, los resultados no serán satisfactorios.

Si se decide escribir un procedimiento con el del Ejemplo 8, una solución posible es emitir explícitamente un X-LOCK en el SELECT para evitar que ninguna otra transacción pueda leer los datos y generar en consecuencia una inconsistencia:

```
SELECT @NuevoMontoOrigen=Monto-@vImporteMover
FROM PresupuestoAnual
WITH (XLOCK)
WHERE . . . . .
```

Los SELECT emiten un S-LOCK (SHARED=COMPARTIDO) lo que permite que la tupla pueda ser leída por otras transacciones, por esa razón se produjo la inconsistencia.

Notas:

- La transacción que obtiene un S-LOCK es la única que , mientras el S-LOCK persiste, puede hacer un X-LOCK sobre la misma tupla o grupo de tuplas.
- En los niveles de aislamiento REPEATABLE READ o SERIALIZABLE el S-LOCK persiste durante toda la transacción, en READ COMMITED se libera al terminar la consulta.

Ejemplo 9

Prevención de LOST UPDATES

```
CREATE PROCEDURE [dbo].[sp_TraspasoPresupuestoV3]
    @vIdAreaOrigen INT,
    @vIdAreaDestino INT,
    @vAnio INT,
    @vImporteMover NUMERIC(12,2) AS
BEGIN
    BEGIN TRY
        BEGIN TRANSACTION
            DECLARE @NuevoMontoOrigen NUMERIC(12,2)
            SELECT @NuevoMontoOrigen=Monto-@vImporteMover FROM PresupuestoAnual WITH (XLOCK)
                WHERE idAreaAdmin=@vIdAreaOrigen and Anio=@vAnio
            WAITFOR DELAY '00:00:20'
            UPDATE PresupuestoAnual
                SET Monto=@NuevoMontoOrigen
                WHERE idAreaAdmin=@vIdAreaOrigen and Anio=@vAnio
            DECLARE @NuevoMontoDestino NUMERIC(12,2)
            SELECT @NuevoMontoDestino=Monto+@vImporteMover FROM PresupuestoAnual WITH (XLOCK)
                WHERE idAreaAdmin=@vIdAreaDestino and Anio=@vAnio
            WAITFOR DELAY '00:00:20'
            UPDATE PresupuestoAnual
                SET Monto=@NuevoMontoDestino
                WHERE idAreaAdmin=@vIdAreaDestino and Anio=@vAnio
        COMMIT
    END TRY
    BEGIN CATCH
        ROLLBACK
        IF ERROR_MESSAGE() like '%CK_PresupuestoMonto%'
            THROW 50001,'¡Monto insuficiente para el traspaso!',1
    END CATCH
END
```

Añada las cláusulas en color verde al procedimiento almacenado.

5.4 Niveles de Aislamiento

-- EJECUTE LOS TRASPASOS SIMULTANEOS

Consola:

```
EXECUTE dbo.sp_TraspasoPresupuestoV3 1,2,2024,10000
```

SSMS:

```
EXECUTE dbo.sp_TraspasoPresupuestoV3 1,3,2024,20000
```

Consola:

-- ESPERE A QUE TERMINEN LAS TRANSACCIONES

```
SELECT dbo.fn_PresupuestoTotal(2024)
```

```
SELECT * FROM vPresupuestoAnual
```

```
GO
```

Ejemplo 10.1

Nivel de Aislamiento READ COMMITTED

```
ALTER PROCEDURE dbo.sp_TraspasoPresupuestoV3
```

```
    @vIdAreaOrigen INT,
```

```
    @vIdAreaDestino INT,
```

```
    @vAnio INT,
```

```
    @vImporteMover NUMERIC(12,2) AS
```

```
BEGIN
```

```
    BEGIN TRY
```

```
        BEGIN TRANSACTION
```

```
        DECLARE @NuevoMontoOrigen NUMERIC(12,2)
```

```
        SELECT @NuevoMontoOrigen=Monto-@vImporteMover FROM PresupuestoAnual WITH (XLOCK)
            WHERE idAreaAdmin=@vIdAreaOrigen and Anio=@vAnio
```

```
        UPDATE PresupuestoAnual
```

```
            SET Monto=@NuevoMontoOrigen
```

```
            WHERE idAreaAdmin=@vIdAreaOrigen and Anio=@vAnio
```

```
        DECLARE @NuevoMontoDestino NUMERIC(12,2)
```

```
        SELECT @NuevoMontoDestino=Monto+@vImporteMover FROM PresupuestoAnual WITH (XLOCK)
            WHERE idAreaAdmin=@vIdAreaDestino and Anio=@vAnio
```

```
        UPDATE PresupuestoAnual
```

```
            SET Monto=@NuevoMontoDestino
```

```
            WHERE idAreaAdmin=@vIdAreaDestino and Anio=@vAnio
```

```
        COMMIT
```

```
    END TRY
```

```
    BEGIN CATCH
```

```
        ROLLBACK
```

```
        IF ERROR_MESSAGE() like '%CK_PresupuestoMonto%'
```

```
        THROW 50001, '¡Monto insuficiente para el traspaso!', 1
```

```
    END CATCH
```

```
END
```

Elimine los
WAITFOR DELAY

Ejemplo 10.2

Nivel de Aislamiento READ COMMITTED

```
CREATE PROCEDURE sp_ReportePresupuesto
    @vTipo VARCHAR(MAX)
AS
BEGIN
```

Crear este
procedimiento

```
    BEGIN TRY
```

Establecer este nivel de aislamiento

```
        BEGIN TRANSACTION
```

```
        SET TRANSACTION ISOLATION LEVEL READ COMMITTED
```

```
        SELECT * FROM vPresupuestoAnual where Tipo=@vTipo
```

```
        WAITFOR DELAY '00:00:30'
```

```
        SELECT SUM(Monto)
```

```
            FROM vPresupuestoAnual
```

```
            where Tipo=@vTipo
```

```
        COMMIT
```

```
    END TRY
```

```
    BEGIN CATCH
```

```
        ROLLBACK
```

```
        DECLARE @vError VARCHAR(MAX)
```

```
        SET @vError=ERROR_MESSAGE()
```

```
        THROW 50001,@vError,1
```

```
    END CATCH
```

```
END
```

Colocar
este
WAITFOR
DELAY

5.4 Niveles de Aislamiento

Nivel de Aislamiento READ COMMITTED

Consola:

```
EXECUTE sp_ReportePresupuesto 'Subdirección'
```

SSMS:

```
EXECUTE dbo.sp_TraspasoPresupuestoV3 1,2,2024,40000
```

Consola:

Espere los 30 segundos de retraso y observará que el total al final no coincide con la información detallada.

El problema estriba en que como ambos resultados se obtienen en dos consultas independientes y el nivel de aislamiento es READ COMMITTED, al haber un traspaso, ambos resultados son inconsistentes.

Ejecute de nuevo el procedimiento

```
EXECUTE sp_ReportePresupuesto 'Subdirección'
```

Ejemplo 11

Nivel de Aislamiento REPEATABLE READ

```
ALTER PROCEDURE sp_ReportePresupuesto
    @vTipo VARCHAR(MAX)
AS
BEGIN
```

Modifique este
procedimiento

Establecer este nivel de aislamiento

```
    BEGIN TRY
        BEGIN TRANSACTION
        SET TRANSACTION ISOLATION LEVEL REPEATABLE READ
        SELECT * FROM vPresupuestoAnual where Tipo=@vTipo
        WAITFOR DELAY '00:00:30'
        SELECT SUM(Monto) FROM vPresupuestoAnual where Tipo=@vTipo
        COMMIT
    END TRY
    BEGIN CATCH
        ROLLBACK
        DECLARE @vError VARCHAR(MAX)
        SET @vError=ERROR_MESSAGE()
        THROW 50001,@vError,1
    END CATCH
END
```

Conservar este WAITFOR DELAY

5.4 Niveles de Aislamiento

Nivel de Aislamiento REPEATABLE READ

Consola:

```
EXECUTE sp_ReportePresupuesto 'Subdirección'
```

SSMS:

```
EXECUTE dbo.sp_TraspasoPresupuestoV3 1,2,2024,25000
```

Consola:

Espere los 30 segundos de retraso y observará que el total al final ahora efectivamente coincide con la información detallada.

Como el nivel de aislamiento es REPEATABLE READ, se obtienen los mismos resultados durante toda la transacción y se mantiene la consistencia.

Ejecute de nuevo el procedimiento

```
EXECUTE sp_ReportePresupuesto 'Subdirección'
```

Ejemplo 12

Nivel de Aislamiento SERIALIZABLE

```
CREATE PROCEDURE sp_AnadirArea
    @vNombre VARCHAR(MAX),
    @vTipo VARCHAR(MAX)
```

Se debe crear este procedimiento

```
AS
```

```
BEGIN
```

```
    BEGIN TRY
```

```
        BEGIN TRANSACTION
```

```
        DECLARE @vNumeroAreas INT
```

```
        SET TRANSACTION ISOLATION LEVEL SERIALIZABLE
```

```
        SELECT @vNumeroAreas=COUNT(*) FROM AreasAdministrativas
```

```
        WHERE idAreaAdmin<=POWER(CAST(2 AS BIGINT),31)
```

```
        INSERT INTO AreasAdministrativas
```

```
        VALUES (@vNombre,@vTipo)
```

```
        WAITFOR DELAY '00:00:30'
```

```
        COMMIT
```

```
    END TRY
```

```
    BEGIN CATCH
```

```
        ROLLBACK
```

```
        DECLARE @vError VARCHAR(MAX)
```

```
        SET @vError=ERROR_MESSAGE()
```

```
        THROW 50001,@vError,1
```

```
    END CATCH
```

```
END
```

Establecer este nivel de aislamiento (pruebe con otro nivel de aislamiento)

Esta consulta bloquea no solo las tuplas que contiene la tabla sino los índices, para evitar que se añada una tupla en otra sesión (el nivel SERIALIZABLE tiene esa capacidad). Como el SELECT emite S-LOCKS, ninguna otra sesión podrá insertar una tupla porque los INSERT emiten un X-LOCK y otra transacción no lo conseguirá debido al S-LOCK emitido por la primera sesión a todos los índices

5.4 Niveles de Aislamiento

Nivel de Aislamiento SERIALIZABLE

Consola:

```
EXECUTE sp_AnadirArea 'Biotecnología', 'Departamento'
```

SSMS:

```
BEGIN TRAN  
INSERT INTO AreasAdministrativas  
VALUES ('Subdirección de TI', 'Subdirección')  
COMMIT  
SELECT * FROM AreasAdministrativas
```

Con otro nivel de aislamiento en el procedimiento almacenado, este insert se ejecuta de inmediato, con SERIALIZABLE entrará en espera

Consola y SSMS:

Espere los 30 segundos de retraso y observará que terminan ambas transacciones, el INSERT en SSMS, se hizo una vez que se efectuó el de Consola.

5.4 Niveles de Aislamiento

Observaciones importantes

- Una transacción siempre emite un bloqueo exclusivo de los datos que modifica y mantiene ese bloqueo hasta que se completa (la transacción), esto es cierto para cualquier nivel de aislamiento excepto **READ UNCOMMITTED** que no emite bloqueos.

5.4 Niveles de Aislamiento

- Para las operaciones de lectura, los niveles de aislamiento definen principalmente el nivel de protección contra los efectos de las modificaciones realizadas por otras transacciones.
- Un nivel de aislamiento más bajo aumenta la capacidad de muchos usuarios para acceder a los datos al mismo tiempo, pero tiene algunos efectos como lecturas sucias o fantasmas con los que hay que ser cuidadoso al procesar los datos.

5.4 Niveles de Aislamiento

Read Uncommitted.

- Hay poco aislamiento presente porque pueden leerse datos no entregados (*Lecturas Sucias*) que es posible que no se guarden en la base de datos.
- Una aplicación difícilmente puede confiar en *Lecturas Sucias*.
- Por las razones anteriores se piensa que READ UNCOMMITTED no es un nivel de aislamiento, sin embargo un argumento contrario es que al haber una transacción de por medio, que puede deshacerse, es un nivel de aislamiento.

5.4 Niveles de Aislamiento

Read Committed.

- Se evitan las *lecturas sucias* ya que los cambios no entregados por otras transacciones no son visibles.
- Garantiza que cualquier dato leído está efectivamente entregado a la BD al momento que se lee.
- Este es el nivel de aislamiento *default* de SQL Server.

5.4 Niveles de Aislamiento

Read Committed.

- Si se hace la lectura de las mismas tuplas múltiples veces dentro de una transacción, se pueden obtener datos diferentes porque el S-LOCK se libera una vez terminado el *SELECT*.
- Este fenómeno se puede presentar porque puede haber modificaciones de las tuplas en otras transacciones. Esto es conocido como LECTURAS NO REPETIBLES.
- Se puede utilizar cuando se requiere conocer en todo momento los datos efectivamente guardados en la BD.

5.4 Niveles de Aislamiento

Repeatable Read.

- Ignora los cambios realizados por otras transacciones, para asegurar *LECTURAS REPETIBLES*.
- Asegura que las tuplas leídas por una transacción no pueden ser modificadas ni eliminadas por otras transacciones hasta que la transacción original haya finalizado porque los S-LOCK persisten hasta el fin de la transacción.
- Es muy útil en la generación de reportes cuyos datos se obtienen con múltiples lecturas, por ejemplo, reportes detallados en una sección y totales o subtotales en otras secciones del reporte.

5.4 Niveles de Aislamiento

Serializable.

- Se evita un fenómeno llamado lecturas fantasma porque se bloquean los índices para evitar que algunas tuplas añadidas aparezcan como lecturas fantasmas en otra sesión.
- *REPEATABLE READ* y *SERIALIZABLE* pueden impactar el rendimiento en sistemas con alta concurrencia debido a la duración más larga de los bloqueos.
- Este nivel de aislamiento es el más fuerte posible.

5.5 Interbloqueos (*DeadLocks*)

- Un *DeadLock* (literalmente “bloqueo muerto” pero traducido como **interbloqueo**) ocurre cuando dos o más transacciones se bloquean mutuamente, cada una esperando que la otra libere las tuplas que necesita para continuar.
- Como resultado, ninguna de las transacciones puede continuar, creando un ciclo de espera infinita que solo puede resolverse si una de las transacciones se cancela (es asesinada según el termino en inglés: *killed*) y se deshace para garantizar la integridad de los datos.
- Este fenómeno se presenta sin importar el nivel de aislamiento presente, excepto READ UNCOMMITTED.

5.5 Interbloqueos (*DeadLocks*)

SQL Server utiliza el siguiente procedimiento para detectar *DeadLocks* y permitir que las transacciones restantes continúen:

- 1. Detección:** SQL Server periódicamente revisa el estado de las transacciones activas para identificar ciclos de espera infinitos.
- 2. Seleccionar una Víctima:** Una vez detectado el *DeadLock*, SQL Server elige una transacción para cancelar. La selección se basa en el costo de revertir cada transacción. SQL Server preferirá cancelar la transacción con el menor costo de retroceso (aquella que ha realizado menos cambios).

5.5 Interbloqueos (*DeadLocks*)

- 3. Cancelar la Transacción:** La transacción seleccionada como víctima es revertida (*rollbacked*). Esto libera los recursos que tenía bloqueados, permitiendo que la otra transacción continúe.
- 4. Notificación:** SQL Server notifica a la aplicación que inició la transacción cancelada. El mensaje de error incluye la frase *deadlock victim*.

Este proceso garantiza que al menos una de las transacciones pueda continuar, rompiendo el ciclo infinito que impedía la ejecución de ambas transacciones.

Ejemplo 13

```
ALTER PROCEDURE dbo.sp_TraspasoPresupuesto
    @vIdAreaOrigen INT,
    @vIdAreaDestino INT,
    @vAnio INT,
    @vImporteMover NUMERIC(12,2) AS
BEGIN
    BEGIN TRY
        BEGIN TRANSACTION
        UPDATE PresupuestoAnual
            SET Monto=Monto-@vImporteMover
            WHERE idAreaAdmin=@vIdAreaOrigen and Anio=@vAnio
        WAITFOR DELAY '00:00:20'
        UPDATE PresupuestoAnual
            SET Monto=Monto+@vImporteMover
            WHERE idAreaAdmin=@vIdAreaDestino and Anio=@vAnio
        WAITFOR DELAY '00:00:20'
        COMMIT
    END TRY
    BEGIN CATCH
        ROLLBACK
        DECLARE @vError VARCHAR(MAX)
        SET @vError=ERROR_MESSAGE()
        IF @vError like '%CK_PresupuestoMonto%'
            THROW 50001, '¡Monto insuficiente para el traspaso!',1
        ELSE
            THROW 50001,@vError,1
    END CATCH
END
```

La diferencia con el procedimiento original solo es el CATCH

5.4 Niveles de Aislamiento

Consola:

```
EXECUTE dbo.sp_TraspasoPresupuesto 3,5,2024,5000
```

Observe que los traspasos son entre las mismas áreas pero a la inversa, lo que ocasiona el **DeadLock**

SSMS:

```
EXECUTE dbo.sp_TraspasoPresupuesto 5,3,2024,7000
```

Consola o SSMS:

Observe que en una de las sesiones se obtiene el siguiente error, indicando que la transacción se canceló:

Transaction (Process ID 52) was deadlocked on Lock resources with another process and has been chosen as the deadlock victim. Rerun the transaction.