UNO. Continuando teoría respecto de Concurrencia y Recuperación ante fallas.

ACLARACION: Para avanzar rápido en lo que queda de este tema, estoy reutilizando el resumen de 2 clases de un semestre anterior, ya que al inicio de cada clase se repasa algo de la anterior, algunos tópicos se mencionan más de 2 veces.

RESUMEN DEL PROBLEMA DE INTERBLOQUE

Es el problema más frecuente y el más importante, debido a que el tamaño de los objetos que se bloquean no necesariamente son filas de tablas, si no mas bien el bloque, página o la palabra que contiene el objeto

Si ocurre, se soluciona victimizando uno de los procesos que intervienen.

Si es el escalamiento del interbloqueo es muy rápido, puede haber una inundación de la memoria y la operación se ralentiza, o en el peor de los casos, la app se cae.

Si las transacciones de BD no están bien programadas puede haber pérdida de datos.

Se previene o reduce utilizando estas buenas prácticas:

- Determina adecuadamente el nivel de aislamiento
- El bloqueo de los objetos siempre el mismo orden
- Evitar el control de flujo dentro de la transacción
- Transacciones lo más pequeñas posibles (en términos de código) y siempre al final del código del SP
- El orden de los objetos debe ser según su importancia de menor a mayor.
- Preprocesar la mayor cantidad de información antes de iniciar la transacción, para reducir carga o peso de la misma.

Las mejores prácticas se evalúan en la 3era tarea, o sea se bajan puntos si la transacción está mal codificada.

Hay otras estrategias de prevención de interbloqueo, que son propias del SABD, no depende del código de programador, no hay tiempo de estudiarlas ….

Respecto de **problemas típicos de la concurrencia**, ya se habló del interbloqueo, otros son **Actualización perdida** y **lectura de datos sucios**. En general es el procesamiento (por ejemplo lectura para un reporte) de muchos registros, al mismo tiempo que otro proceso los está actualizándolos.

Hay una variante llamada registro fantasma, cuando un proceso analiza un conjunto de información y al mismo tiempo otro proceso inserta objetos en ese conjunto. El primer proceso ignora la existencia de los registros insertados por el 2do (registro fantasma).

Proceso 1: sumariza los saldos de las cuentas de 2,000,000 cuentas. Y tarda 10 segundos iniciando en tiempo T1.

Select sum(C.saldo)

From dbo.Cuentas,

Order by C.Id

Proceso 2. Actualiza el saldo de la cuenta 123 en tiempo T1+1 segundo, le aplica un depósito de 100, ya el Proceso 1 ha sumarizado la cuenta 123.

Proceso 1 Al final arroja un resultado R que es la sumarización de todos los saldos, que no es consistente pues es R +100, o sea que el resultado "perdió la actualización". Proceso 2 ensució la información sumarizada por Proceso 1.

C1.Sald=50, C2.Saldo=75, c3.Saldo=40

Proceso 1	Proceso 2
P0 SumaTotal=0	
P1 SumaTotal= SumaTotal+c1.Saldo	
P2 SumaTotal= SumaTotal+c2.Saldo	C1.Saldo=C1.Saldo+100
P3 SumaTotal= SumaTotal+c3.Saldo	
P4 Print SumaTotal (165) debió ser 265	

¿Como resolver este problema?

Una posibilidad es que el proceso 1 bloquee la información que va a leer.

```
Select sum(C.saldo)
From dbo.Cuentas with (XLOCK)
Order by C.Id
```

XLock produce un bloqueo exclusivo, ningún otro proceso accede a la información hasta que el proceso 1 termina, el proceso 2 tendrá que esperar.

Se puede ser más radical:

```
Select sum(C.saldo)
From dbo.Cuentas with (TABLOCK, XLOCK)    -- bloquea toda la tabla
Order by C.Id
```

No es buena práctica, que el programador explícitamente especifique los bloqueos (a través de hints), a menos que sea un programador experto. Excepto se hagan bloqueos NOLOCK en tablas catálogo. Actualización de catálogos es poco frecuente, y no deben actualizarse mediante interfaz de usuario, sino a través de scripts.

El bloqueo, si no hay hint, lo establece el SABD dentro de una transacción, el programador puede influir sobre el mismo según el nivel de aislamiento. El dba puede definir el nivel de aislamiento por default, el default al instalar el SABD es read committed (un objeto es leído solo si ha sido commiteado, si dos transaccion accden a mis objeto, una de ellas espera mientras la otra hace commit).

Begin tran

.....

Update C ~~(rowlock, updlock)~~ – la Buena practica es no escribir hint

C1.Saldo=C1.Saldo+100

Where <P>

.....

Commit tran

Precondición: la transacción de la BD corriendo sola (de manera no concurrente), siempre es consistente. Las fallas (referidas a la recuperación), son del contexto (del SO, físicas – del hardware: disco duro, memoria, ambientales por caída de rayería o inundación, -o del contexto de una corrida concurrente – la transacción es víctima para resolver un problema de consistencia o para respetar un protocolo que previene un error de consistencia).

Comentario al margen: toda la teoría sobre este tema ha sido impactada por la tecnología de las nuevas memorias SSD (son mas rápidas, almacenan mas y duran mas pues el contacto físico es mínimo) y por el almacenamiento en la nube que permiten habilitar redundancia de tablas (a nivel vertical), en servidores espejo, en la instancia en espejo se hace la consulta, la instancia base se hace la actualización, la sincronización es en tiempo real. La nube, que se implementa en granjas de servidores permite el particionamiento horizontal, de manera que una tabla no está en el único servidor, permitiendo proceso en paralelo, lo cual reduce los umbrales de tiempo de ejecución. Un servidor en la nube (en un co-location en Alaska, a 50 metros bajo tierra no le llegan los rayos, ni se inunda, etc.). Sin embargo, la teoría sigue vigente, pues también ha crecido el volumen de las operaciones.

Problemas que se derivan de una ejecución concurrente: interbloqueo, actualización perdida y lectura sucia, y otros que son variantes de los 2 últimos.

En la párrafos anteriores el problema que se ilustró al final, en realidad es de Lectura sucia.

P1 “largo”, sumariza los saldos de N cuentas, P2 es un proceso “corto” que cambia el saldo de una cuenta (alguna que se sumariza en P1), La actualización de P2 a la cuenta c1, sucede poco después de que P1 la sumariza, entonces P1 leyó el valor anterior y no el actual, entonces P1 leyó un dato “sucio”, por lo tanto, su sumarización no es correcta.

Para evitar problemas de lectura sucias, o P1 (que solo lee) bloquea las N cuentas que va a leer, así cualquier transacción que inicie deberá esperar, o P1 es una transacción con nivel de aislamiento que solo permite que sus objeten se acceden solo si han sido “commiteados”, en ese caso la transacción P2 tendría que esperar. O las 2, p1 bloquea y P2 es transacción con nivel de aislamiento read committed (SET TRANSACTION ISOLATION LEVEL READ COMMITTED, que se escribe una línea antes de BEGIN TRANSACTION). En cualquier caso, el resultado será una serialización de las transacciones, alguna inicia antes que la otra (que debe esperar), o si por alguna razón corren concurrentemente, alguna será implícitamente “rollbackeada” para que inicie nuevamente pero después.

El problema de **actualización perdida** es que hay dos procesos corriendo, P1 y P2, P1 actualiza un objeto, y poco después P2 también lo hace, el resultado final es como si P1 nunca hubiera corrido.

Paso en el tiempo	P1	P2
t1	Read (a)	
t2	a=a+5 (en memoria)	Read(a) lectura de BD física
t3	Write (a) (escritura en BD fisica)	… aquí se ensucia el valor de a
t4		a=a+7
t5	Fin	
t6		Write(a)
t7		Fin

P2 reescribe el objeto a, de tal manera que se pierde la actualización de a hecha por P1 en t3.

Para resolver, debemos transformar P1 y P2, en transacciones de base de datos Tr1, Tr2. Si ambas son transacciones, en t2, Tr2 tendrá que esperar hasta que Tr1 llegue a commit, de manera que cuando Tr2 deje de esperar y continua, va a leer un valor de a fresco (no sucio) de a. El uso de transacciones obliga a una serialización.

Otro problema que es una variante del de “lectura sucia”, es el del “registro fantasma”. Un proceso inserta un registro, que debe ser contado o tomado en cuenta por otro proceso que ignora esa inserción.

Otra forma de evitar problemas de concurrencia es establecer bloqueos explícitos, las buenas prácticas dicen que solo se debe hacer si se es un programador de BD con experiencia y se conocen la sensibilidad de los datos a problemas con bloqueos. La idea detrás de esta buena práctica es que el SABD siempre conoce mejor el contexto que el programador y tomará mejores decisiones, entonces el programador lo que hará es que especifica el nivel de aislamiento y el SABD tomara las decisiones respecto a los bloqueos.

NOLOCK

El NOLOCK permite lecturas sucias, o sea no bloquea, no protege la informacion, el registro puede estar siendo actualizado en ese momento. Favorece la concurrencia, pero puede producir inconsistencias.

```
SELECT C.TipoCuenta, C.FechaCreacion
FROM dbo.Cuentas  C (NOLOCK)
WHERE C.id=@inCuentaId
```

Lo anterior es aceptable, debido a que es infrecuente el cambio de la FechaCreacion de La cuenta o el tipo de cueenta.

Se debe utilizar solo si la información a leer no es tan sensible (no habrá demandas o riesgo de daños a la integridad de las personas).

Si se estuviera leyendo el saldo de una cuenta, un NOLOCK es inaceptable.

```
SELECT C.TipoCuenta, C.FechaCreacion, C.Saldo
FROM dbo.Cuentas  C (NOLOCK)
WHERE C.id=@inCuentaId
```

Hay que recordar que los bloqueos se mantienen hasta que termine la conexión, o sea hasta finaliza el código que contiene el bloqueo. Eso explica el pq es importantísimo que la transacción inicie lo mas tarde posible, que esta al final del código, y que sea lo mas corta posible.

Opciones: no poner nada (el SABD decide el bloqueo que aplica), o poner explícitamente algún bloqueo.

La recomendación es no poner nada, ejemplo

```
SELECT C.Nombre, C.FechaNacimiento, C.Saldo
FROM dbo.Cuentas  C
WHERE C.id=@inCuentaId
```

(SHRLCK)

Permite que otros procesos lean el objeto, PERO no lo pueden actualizar hasta que proceso que tiene el shrlock termine. Protege los objetos de ser actualizados, aunque permite compartirlos a través de lectura.

```
SELECT C.Nombre, C.FechaNacimiento, C.Saldo
FROM dbo.Cuentas  C  (shrlock)
WHERE C.id=@inCuentaId
```

El sistema protege que el objeto ‘cuenta’ (así como los registros cercanos =@inCuentaId), que sea actualizado por otros procesos, pero SI permite que otros procesos accedan al objeto para lectura, solamente.

ALGO IMPORTANTE, es que los bloqueos (que se especifican a través de los hints) se mantienen hasta el final del proceso, y si están dentro de una transacción se mantienen hasta commit o rollback.

El shared lock, favorece la concurrencia, y previene las lecturas sucias. Muchos procesos pueden tener (compartir) shrlock sobre mismos objetos.

(UPDLCK).

El updlck, se usa cuando está la posibilidad de que el objeto que se está leyendo vaya a ser actualizado. Es compatible con shared locks que tengan otros procesos, pero el Updlock solo es concedido **a un único proceso** (si un 2do proceso solicita un updlock sobre el mismo objeto, y el 1ero no ha terminado, el 2do queda en espera). El updlock se promueve a Xlock en tiempo de actualización.

```
SELECT C.TipoCuenta, C.FechaCreacion, C.Saldo
FROM dbo.Cuentas  C  (Updlock)
WHERE C.id=@inCuentaId

...

....

Update dbo.Cuenta                          -- en este momento el bloqueo se promueve a XLock
Set C.Saldo=C.Saldo+10
WHERE C.id=@inCuentaId

....
```

En el momento que se promueve a Xlock, otros procesos que intentan leer (shrlock) o actualizar a través de updlock y xlock, tendrán que esperar.

Cuando el bloqueo se promueve a XLock, si otros procesos tienen un shrlock, sobre el objeto, el proceso que "pide" el Xlock tiene que esperar que los otros procesos terminen.

XLOCK

Si concedido, el acceso del proceso al objeto es exclusivo, ningún otro proceso puede accederlo ni de lectura ni de actualización (a menos que estén utilizando el NOLOCK). El XLOCK es implícito en toda actualización (delete, update o insert).

```
SELECT C.Nombre, C.FechaNacimiento, C.Saldo
FROM dbo.Cuentas  C  (XLock)
WHERE C.id=@inCuentaId
```

Todos los procesos que quieran leer este objeto deberán esperar a que el proceso que tiene el xlock termine.

Hay otros tipos bloqueos: el intencional, el hold lock, etc, etc. La idea es no usarlos a menos que el sistema, por volumen o antigüedad sea necesario optimizarlo y el SABD (solito) no lo mejora

ESPECIAL ATENCION A UN TIPO DE BLOQUEO LLAMADO **READPAST,** no lo explicamos por falta de tiempo, sin embargo, es muy útil para implementar colas en procesamiento de BD.

Hay especificación de bloqueos (hints) que tienen que ver con **la granularidad del bloqueo**, hay 3 niveles:

Tablock (se bloquea toda la tabla), sector o página (default), rowlock (registro).

Tablock es algo muy radical, el rowlock es muy útil y cambió el paradigma sobre como programar en SQL, de tal forma que, para reducir problemas de concurrencia, es mejor iterar que hacer actualizaciones masivas o procesos en lote (procesos batch) – recordar tarea escrita.

Ejemplo: ”Insertar un bono de 200$ a todas las cuentas cuyo fecha de creación corresponde a la fecha de proceso”.

Tenemos 4 maneras de hacerlo.

F0. Sin bloqueos, ni uso de transacción de BD. Con peligro de que otros procesos lean sucio, es inaceptable.

F1. Actualización masiva, como se ha hecho por muchos anos en SQL antes del advenimineto del rowlock

```
BEGIN TRY
    SET TRANSACTION ISOLATION LEVEL READ COMMITTED
    BEGIN TRAN tBonoCumple
        INSERT db.Movimientos (IdCuenta, idTipoMov, Monto, FechaOperacion)
        SELECT C.Id, 1, 200, @inFechaOperacion
        FROM dbo.Cuenta C
        WHERE dbo.CumpleAnosCuenta (C.FechaCreacion, @inFechaproceso)=1

        UPDATE dbo.Cuenta C
        SET Saldo=Saldo+200
        WHERE dbo.CumpleAnosCuenta (C.FechaCreacion, @inFechaproceso)=1

    COMMIT TRAN tBonoCumple

    SET @OutResponseCode=0;

END TRY

BEGIN CATCH

    IF @@TRANCOUNT>0

        ROLLBACK tBonocumple

        SET @OutResponseCode=50005;

END CATCTH
```

F3. Usando RowLOCK. La otra opción es usando rowlock, e iterando. (la que el profe utiliza actualmente).

```
BEGIN TRY

    DECLARE @CuentasCumple TABLE (Sec INT IDENTITY(1, 1), IdCuenta INT);

    DECLARE @lo INT=1, @hi INT, @idCuenta INT;

    INSERT @CuentasCumple(idCuenta)

    SELECT C.Id FROM dbo.Cuentas C
    WHERE dbo.CumpleAnosCuenta (C.FechaCreacion, @inFechaproceso)=1

    SELECT @hi=MAX(Sec) FROM @CuentasCumple;

    SET TRANSACTION ISOLATION LEVEL READ COMMITTED
    BEGIN TRAN tBonoCumple

    WHILE @lo<=@hi

    BEGIN

        Select @IdCuenta=C.IdCuenta FROM @CuentaCumple C WHERE C.sec=@lo

        INSERT dbo.Movimientos (IdCuenta, idTipoMov, Monto, FechaOperacion)
        values( @idCuenta, 1, 200, @inFechaOperacion)

        UPDATE dbo.Cuenta C (ROWLOCK)  -- EL BLOQUEO EX IMPLICITAMENTE XLOCK
        SET Saldo=Saldo+200
        WHERE id=@idCuenta

        SET @lo=@lo+1

    END

    COMMIT TRAN tBonoCumple

    SET @OutResponseCode=0;

END TRY

BEGIN CATCH

    IF @@TRANCOUNT>0

        ROLLBACK tBonocumple

        SET @OutResponseCode=50005;

END CACTH
```

F2 VS F3. Si las transacciones corren solas F2 es más eficiente, pues inserta y actualiza masivamene.

Si corren concurrente con otras transacciones, F2 tiene mayor probabilidad de bloquear objetos, o sea hacer que las otras transacciones esperen, o se generen deadlocks u otros problemas, pues su granularidad es alta. F3 será más lenta, pero más segura, porque su granularidad es pequeña.

Granularidad es el tamaño del bloqueo, en F2 se bloquean masivamente un monto de cuenta, que si se encuentran en forma fragmentada en el disco, será un bloqueo grande y distribuido, la probabilidad de que haya un interbloqueo es mayor. F3 va bloqueando de 1 en 1, cada registro, y cada bloqueo es pequeño (rowlock), la probabilidad de un interbloqueo es menor.

Analizando las maneras de implementar, el proceso masivo (aplicándolo a muchas instancias en el proceso) que le da 200 colones de bono a las cuentas de ahorro que cumplen anos en una fecha de operación (respecto la fecha de creación de la cuenta).

F0: sin uso de transacción de base de datos, esto es inaceptable pues permitiría lecturas sucias sobre el saldo de la cuenta y el proceso no es atómico.

F2: se usa una transacción de BD, dentro de ella se usa sql masivo (insert masivo, de muchas instancias; o update masivo, de muchas instancias). El proceso corriendo solo o con poca actividad en la BD, correrá muy rápido; pero corriendo concurrente y con mucha actividad, provocará bloqueos o esperas en los demás procesos, pues la granularidad de sus bloqueos será grande y dispersa. Hay mayor probabilidad de que existan deadlocks.

F3: usando rowlock, todo el proceso es atómico, dentro de la transacción de la BD se itera por cada cuenta que cumple años, se inserta el movimiento y se actualiza el saldo, usando rowlock. Esto esta bien, cuando las actualizaciones por cada instancia son pocos o simples, en este caso solo se tocan 2 entidades: cuenta y movimiento. No es tan conveniente cuando las actualizaciones en cada iteración son complejas.

F4. Los cálculos para el procesamiento de una instancia, son complicados, largos y actualizan muchas entidades (como es el caso de procesar una asistencia en la tarea programada de las planillas), si el proceso para todas las instancias se encierra en una transacción de BD, esta transacción seria inmensa y dentro de ella tendríamos control del flujo (ciclos, if then else, llamadas a funciones y SPs), lo cual provocara que la transacciones es tan grande que va a bloquear grandes porciones de la BD haciendo el proceso pesadísimo.

Para F4, lo que se propone que la transacción de la BD solo se aplique para las actualizaciones de una sola instancia (solo para el procesamiento de una marca de asistencia) y usando rowlock, y que para asegurar la atomicidad del procesamiento masivo, se haga programáticamente, o sea que lo haga el programador, utilizando tablas auxiliares y controles para asegurar que el proceso masivo corre completo para todas las instancias SOLO UNA VEZ, y para cada instancia corre completo y solo una vez.

La opción F4 que ya comentamos antes en el curso, es que la atomicidad de todo el proceso se haga programáticamente, de manera que solo se bloquea una instancia dentro del while que itera sobre todos los objetos.

NIVELES DE AISLAMIENTO.

Respecto de la I del ACID, que son los niveles de aislamiento, estos se proponen antes del begin transaction, son

SET TRANSACTION LEVEL <nivel de aislamiento>

<niveles de aislamiento> :: =

Read uncommitted ; la transacción puede leer datos sucios, inaceptable para proceso de datos sensibles, susceptibles a demanda o riesgo de la vida humana.

Repetable read: si los datos de la transaccion se ensucian, la transaccion hace rollbak y renicia de manera para leer datos frescos.

Read committed: transaccion protege de lectura de datos sucios

Data snapshots: requiere comentario largo ....

Serializable: la transacccion corre sola. Todas las demás esperan.

D de Durable, respecto del acrónimo ACID.

Las transacciones de la BD son durables, porque una vez que la transacción llegó a commit, queda en la BD aunque haya una falla inmediatamente después del commit, talque impida que alguna actualización llegue a la BD físico, entonces el proceso de recuperación de la falla debe asegurar que esas actualizaciones SI lleguen a la BD física.

Cuando se hace commit no necesariamente los datos actualizados van a la BD física, puede se que queden en buffers.

La falla es generada por el contexto, y no es una falla en el código, que suponemos que el código es correcto si corre solo (no concurrentemente), la falla puede ser: fallo de energía, pulga del sistema operativo, falla del dispositivo, victimización del proceso debido a interbloqueo o aplicación de algún protocolo de manejo de concurrencia (que no podremos estudiar por falta de tiempo).

LOS COMMITS ACTUALIZAN LOS BUFFERS, LAS ACTUALIZACIONES SE REPLICAN A LA BD FISICA CUANDO SE EJECUTA UN **CHECKPOINT** O CUANDO SE PRODUCE UN REEMPLAZO DE PAGINAS EN LOS BUFFERS.

Un buffer se implementa como un arreglo de paginas en memoria ram, lecturas de BD física no se hacen directamente, se hacen a través de buffers, actualizaciones se hacen el buffer. El uso de buffer provee independencia entre el SO y los dispositivos, cambios en los dispositivos implican un cambio en le driver, el cambio es transparente para el SO (sistema operativo). Cuando un buffer se llena, al hacer una lectura debe buscarse un espacio libre en buffer, al no encontrarlo, busca l

página más vieja actualizada la replica a la BD física y se esa manera ‘se hace el campo’ para que la nueva lectura se guarde en el buffer.

Cuando se corre CHECKPOINT; el SABD hace una pausa en todos los queries corriendo y replica todas las actualizaciones en Buffers a la BD física. Es estatuto puede se explicito por el programador o dba, o es implico, pues el SABD lo corre cada x min.

Anteriormente, mencionamos que la BD, en su implementación física se compone de 2 archivos, el de datos (en MSQL con extensión .mdf) y el de bitácora (en MSQL con extensión .ldf). Ambos tienen buffers independientes, aunque el buffer del .ldf tienen prioridad, es más rápido y usualmente corre en memoria cache. Cuando hay falla general, el hardware del servidor tiene una batería que hace que los últimos momentos se usan para llevar al archivo físico, todo lo que esta en la bitácora, de manera que, aunque el .mdf no se actualice, el .ldf sí quedara actualizado.

Nota al margen

Este contexto es diferente en servidores en la nube, son granjas de servidores, hay mucha redundancia de la fuente de energía, hay redundancia de datos de tal manera que la BD esta redundante y en espejo en muchos servidores y en diferentes localizaciones.

Fin Nota al margen.

La bitácora, contiene un registro de todos los eventos en la BD, ya sea de accesos de usuarios, corridas de sentencias sql o objetos programáticos (SPs, funciones, vistas, triggers, etc), y respecto de las transacciones de BD, marca el inicio (Begin tran) y fin de las mismas (Commit y Rollback), así como el salvado de 3 tipos de información para objeto que este siendo actualizado dentro de una transacción a través de delete, update o insert:

Registro de bitácora tipo do: la especificación de la operación que se realiza, la cual equivale al estatuto SQL que se ejecuta para una fila especifica.

Archivo de bitácora .ldf.

La bitácora, contiene un registro de todos los eventos en la BD, ya sea de accesos de usuarios, corridas de sentencias sql u objetos programáticos (SPs, funciones, vistas, triggers, etc), y respecto de las transacciones de BD, marca el inicio (Begin tran) y fin de las mismas (Commit y Rollback), así como el salvado de 3 tipos de información para fila de tabla que este siendo actualizado dentro de una transacción:

Registro de bitácora:

Tipo do: la especificación de la operación que se realiza, la cual equivale al estatuto SQL que se ejecuta para una fila específica, estos registros son necesarios para poder re-ejecutarlos en caso de que la transacción se reinicie. Son el insert, delete o update tal como son corridos. Cada vez que se ejecutan, lo hacen en un contexto diferente.

Tipo Undo: contiene, para todas las filas que están siendo modificadas por un estatuto SQL, los valores de los atributos antes de su ejecución. Cuando se ejecuta un Rollback, estos registros

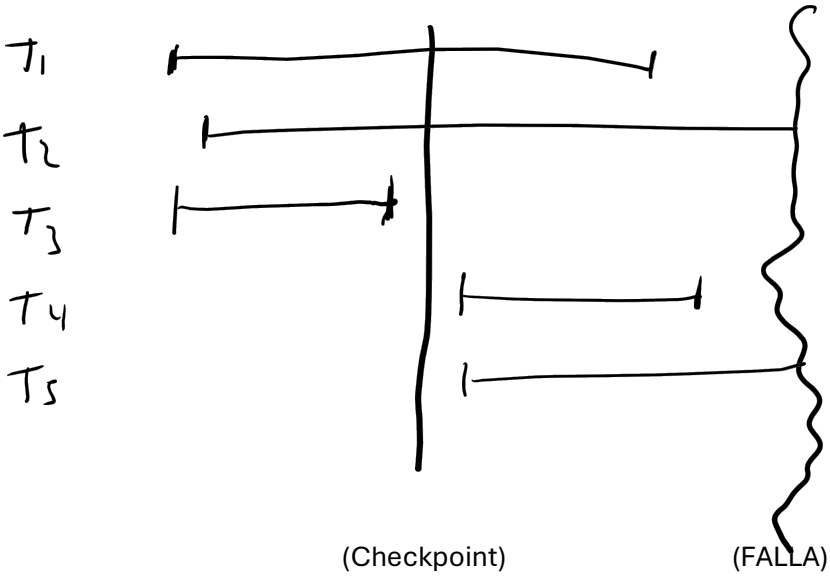
son usados para restaurar los valores antes, y asegurar el “nada” de la A de ACID (Atómico-”Todo o Nada”).

Tipo Rdo: contiene, para todas las filas que están siendo modificadas por un estatuto SQL, los valores de los atributos después de su ejecución. Cuando una transacción de BD se recupera después de una falla, se restauran en la BD física los valores tal como quedaron después de la ejecución de los estatutos para asegurar el “todo” de la A de ACID y la D.

El **checkpoint**, es una operación implícita del SABD (aunque el sa o dbo la pueden correr explícitamente), cada cierta cantidad de minutos (por ejemplo, cada 5 minutos), el cual todas las actualizaciones que están en buffer de datos se llevan a la BD físico, de tal forma que el proceso de recuperación de datos se realiza desde o hasta el último checkpoint. Algunos SABD permiten que el programador haga checkpoints explícitos.

Un checkpoint, hará que todas las transacciones ejecutando hagan una pausa, y que todas las actualizaciones en el buffer de datos se lleven a la BD fisica. Checkpoints demasiado frecuentes generar una disrupción en la operación del sistema.

Se dan 5 situaciones posibles respecto de una falla que se produce después de un checkpoint, T1 hasta T5 son transacciones de BD.



Como se recupera el SABD después de falla

T1: undo hasta ultimo checkpoint y luego redo usando registro del .ldf

T2: undo hasta inicio de la transaccion y se ejecutan los do

T3: no se hace nada

T4: se redo para los cambios que no llegaron a la BD física

T5 idem T3

Despues de checkpoint y antes de la falla, las actualizaciones en BD física SOLO se hicieron debido a reemplazo de páginas en los buffers de salida.

T1: Transacción inicio antes del checkpoint, y llega a commit antes de la falla.

Las actualizaciones llegaron a commit, pero no necesariamente están en la BD física, pues talvez quedaron en buffer de datos.

Acciones en proceso de recuperación:

El SABD, usando la bitácora regresa al punto de checkpoint, avanza sobre los registros do y redo, respecto de los redo chequea si los mismos están reflejados en la BD física, si sí, no hace nada, y si no, replica el redo en la BD física.

T2: La transacción inicio antes del checkpoint, entonces algunas actualizaciones quedaron en la BD fisica, sin embargo, la transacción no terminó pues hubo falla antes de llegar commit. El SABD debe deshacer todos los cambios realizados (de esa manera garantiza el “nada”), y debe reiniciar la ejecución de la transacción, de manera transparente al usuario.

Acciones en proceso de recuperación:

Debe ir hasta el inicio de transacción, usando los registros de la bitácora, y sobre aquellos que son previos al checkpoint, debe aplicarles el UNDO, de manera que los registros queden con los valores antes del inicio de la transacción. Para los registros posteriores al checkpoint, debe chequear si los registros DO quedaron en la BD física, comparándolos con lo registros Redo, si sí, debe deshacer la actualización restaurando los valores en los registros undo, si no, no hace nada.

T3: Inicio antes del checkpoint y llego a commit antes del checkpoint.

Como llego a commit antes del checkpoint, es seguro que los datos están en la BD fisica, por lo tanto no se hace nada.

T4: Transacción inicio después del checkpoint y termino antes de la falla.

El proceso de recuperación debe asegurar el Todo, y que las actualizaciones quedan en la BD física. El problema es que no se sabe que quedo y que no en la BD física.

Acciones en proceso de recuperación:

Puesto que llego a commit, estamos seguros de que la bitácora contiene todos los registros redo de la transacción. Inspecciona todos los registros en orden secuencial desde el inicio de la transacción, compara los valores de los registros redo con los valores en la BD fisica, si la actualización no se hizo, pues la realiza.

T5: La transacción inicio después del checkpoint, y no termino.

Acciones en proceso de recuperación:

Debe deshacer todas las actualizaciones que si llegaron a la BD físico, por razón de reemplazo de paginas en el buffer de datos, y reiniciar la ejecución de la transacción desde cero.

El SABD debe asegurar la idempotencia de la ejecución de las transacciones, pues una transacción puede ser que falle varias veces (fallas del contexto) y debe re-ejecutarse varias veces, cada vez deberá hacer undo de actualizaciones que no llegaron a commit, y el resultado final debe ser como si transacción solo hubiera corrido UNA sola vez. Ello implica que si la transacción hace lecturas o salidas a dispositivos externos (ej: una interfaz de usuario), debe asegurarse, que para el usuario, que las operaciones de entrada (lectura) y salida (ejemplo: un print), solo se realiza una sola vez. EL SABD debe poner una cola todos los mensaje de entrada, tal que si la transacción se reinicia varias veces, hará la lectura desde la cola y NO desde la interacción con el usuario. Igual, las salidas quedan en una cola, de manera que solo se imprimen UNA sola vez y hasta a la llegada del commit.

Hacer lecturas, desde la interfaz de usuario, dentro de una transacción de BD, es NO NO, nunca hay que hacerlo, y obviamente es prohibido en este curso, donde de todas maneras no se puede hacer debido a que es prohibido ejecutar SQL desde capa lógica a menos que sea la ejecución de un SP.

Lo que no debe hacerse en capa lógica. (esto es un NO NO)

```
ConnexionDB.ExecSQL("Begin transaction");
```

Read (a); -- esta lecturas debe hacerse una sola vez ...

Hago cosas en capa lógica ....

ConnexionDB.ExecSQL("insert into Tabla (atributo) values (a);")

.....

ConnexionDB.ExecSQL("Commit transaction");

Lecturas dentro de una transacción de BD no deben hacerse nunca.

Salidas, tal como prints o datasets son aceptables mientras se este depurando ... por ejemplo usando prints dentro de una transacción de BD, sin embargo, la prueba que hizo el profe los prints salen a la interfaz del usuario, incluso después de un rollback explicito. Tal vez porque son prints al Management no al usuario final.

Lo que se quiere evitar, es que, por ejemplo, un cliente de banco llegue al cajero automático, y que la transacción de BD corra varias veces, pues fue reiniciada, y que solicite al cliente mas de una vez el "ingrese el monto que desea retirar" o que imprima varias veces "Dinero entregado, gracias por su preferencia".

La I, del ACID, son los niveles de aislamiento: Nolock o no uso de transacción, con peligro de uso de datos sucios o de actualizaciones perdidas, el read uncomitted que permite lecturas sucias, el read committed que no permite lecturas sucias, el readpast que si un valor leído se ensucia, la transacción se reinicia, y el serializable, cuando la transacción correra sola para evitar inconsistencias, aunque anula la concurrencia.

La C del ACID, asegura que la transacción siempre será consistente al final de un proceso concurrente, para ellos el criterio de corrección es que la ejecución debe ser equivalente a alguna de las posibles ejecuciones seriales.

Hoy en la tarde envió notas previas actualizadas con quiz del 12 de nov, quiz por las asistencia a clases, y quiz de hoy.

Estoy debiendo la rubrica de la 3ra tarea y temario para el examen.

Nos vemos en el taller, de hoy en 8.