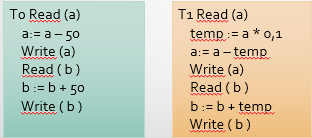
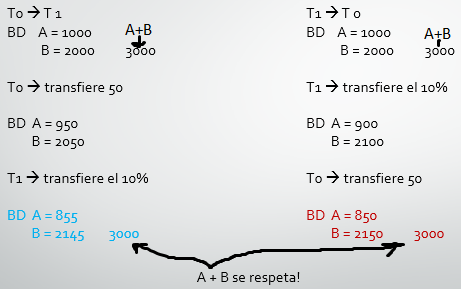
**Entornos concurrentes**

*Entorno centralizado*

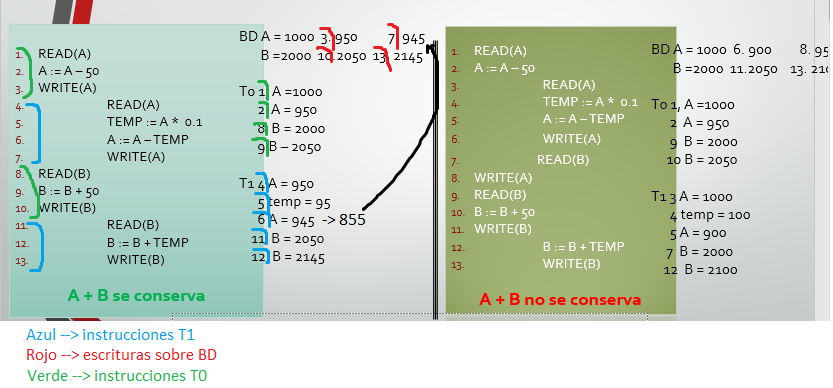
* Varias transacciones ejecutándose simultáneamente compartiendo recursos (el recurso es la BD).
* Deben evitarse los mismos problemas de consistencia de datos.
* Transacciones correctas (ocurriendo al mismo tiempo) en ambientes concurrentes pueden llevar a fallos y destruir la consistencia de los datos.

*Seriabilidad*

* El concepto de seriabilidad es aquel que garantiza la consistencia de la BD después de que dos transacciones se ejecuten sobre los mismos datos.
* 
* Al resolver primero T0 o T1, o T1 y T0 se respeta que A+B es igual para ambos resultados.
* Ahora, T0 T1 <> T1 T0.
* ¿Qué es A+B?, la suma de ambos saldos debe ser igual a X valor sin importar el orden en que se ejecuten las transacciones
* 
* A pesar de que A y B sean distintos en ambos ordenes de ejecución, no me interesa eso, me interesa ver que A+B es igual para ambos casos.
* Si estoy en una ejecución sin fallos, al aplicar una ejecución secuencial de dos transacciones, la consistencia se mantiene.
* T0 y T1, o, T1 y T0 son dos planificaciones diferentes. Ver abajo.

*Planificación: secuencia de ejecución de transacciones*

* La planificación involucra todas las instrucciones de las transacciones.
* Conservan el orden de ejecución de las mismas.
* Un conjunto de m transacciones generan m! planificaciones en serie.
  + M es la cantidad de transacciones que tengo.
* La ejecución concurrente NO necesita una planificación en serie.
  + Fácil, la concurrencia no tiene nada que ver con ejecutar las transacciones en un orden (1 x 1, como si fuera monousuario).
  + Ejecutar en serie las transacciones es una locura ineficiente.
  + Ejecutando concurrentemente se logran respuestas más rápidas.
  + Hay ejecuciones concurrentes que sirven y otras que no sirven (dejan inconsistencia)



* A-temp debería ser 855. Solo así asegurás la consistencia de A+B = 3000. Se lo fumó Bertone al error xd.

*Conclusiones*

* El programa debe conservar la consistencia.
* La inconsistencia temporal (cuando una transacción está realizando algo y viene otra transacción a perjudicar el trabajo propio, ejemplo: T0 y T1 en el ejemplo anterior “A+B no se conserva”, específicamente entre instrucciones 2 y 6) puede ser causa de inconsistencia en planificaciones en paralelo.
* Una planificación concurrente debe equivaler a una planificación en serie, si demostramos que una planificación concurrente equivale a una planificación serie entonces demostramos que el resultado final va a ser el correcto y la consistencia no se va a ver perjudicada (ya que las planificaciones en serie son aquellas que no fallan).
* Solo las instrucciones READ y WRITE son importantes, hay que considerarlas.
  + Las operaciones ajenas a esas dos instrucciones son operaciones locales y no nos interesan para nuestro análisis.

*Conflicto en planificaciones serializables*

* Inst1, Inst2, dos instrucciones de T1 y T2.
  + Solo son read o write las que nos importan ya que actúan sobre la BD.
  + Si operan sobre datos DISTINTOS, NO hay conflicto.
  + Si operan sobre el mismo dato, Inst1 e Inst2 estarán en conflicto si actúan sobre el mismo dato y AL MENOS UNA instrucción es un WRITE.
    - Inst1 = READ(Q) = Inst2 🡪 no importa el orden de ejecución.
    - Inst1 = READ(Q), Inst2 = WRITE(Q) 🡪 depende el orden de ejecución (Inst1 leerá valores distintos en base al orden), provoca conflicto.
    - Inst1 = WRITE(Q), Inst2 = READ(Q) 🡪 depende el orden de ejecución (Inst2 leerá valores distintos en base al orden), provoca conflicto.
    - Inst1 = WRITE(Q) = Inst2 🡪 depende el estado final de la BD, queda la ultima escritura, provoca conflicto.
    - Cuando hay conflicto el orden de ejecución será importantisimo.

*Definiciones*

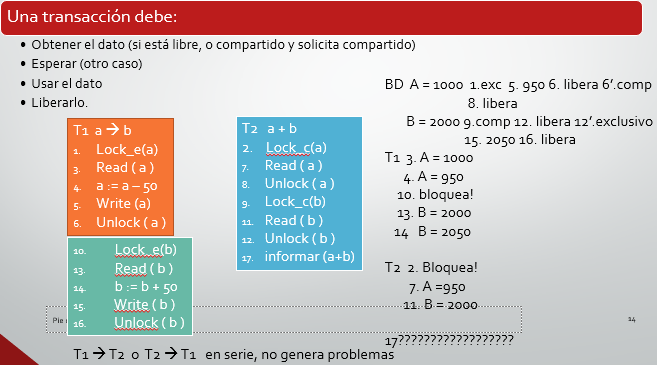
* Una planificación S se transforma en una S’ mediante intercambio de instrucciones no conflictivas (operan sobre datos diferentes o entre dos READ), entonces S y S’ son equivalentes en cuanto a conflictos.
* Esto significa que si:
  + S’ es consistente, S también lo será.
  + S’ es inconsistente, S también será inconsistente.
* Si S’ es serializable en conflictos si existe una planificación S tal que son equivalentes en cuanto a conflictos y S es una planificación serie.
  + Si una planificación S’ concurrente es válida, entonces puedo demostrar que esa planificación concurrente equivale a una planificación serie.

**Control de Concurrencia**

*Métodos de control de la concurrencia:*

* Mecanismo que controla que las planificaciones inválidas concurrentes NO se ejecuten.
* Aseguran que la ejecución simultánea de dos o más transacciones no conflictuen entre sí.
* Tenemos dos métodos.

**Bloqueo: primer método de control**

* Compartido Lock\_c(dato) (solo para lectura): un dato bloqueado de forma compartida, varias transacciones pueden acceder al dato pero SOLO para leerlo.
* Exclusivo Lock\_e(dato) (lectura/escritura): una transacción necesita escribir un dato, por lo que bloquea exclusivamente el dato, y el bloqueo exclusivo me deja cambiar el dato y no permite que OTRA transacción lea el dato en pleno cambio (evita conflictos).
* Las transacciones piden lo que necesitan.
* Los bloqueos pueden ser compatibles y existir simultáneamente (compartidos).
* En una BD bloquear significa pedir el dato que necesito (bloquear), usarlo, y liberarlo (desbloquear).
* 
  + Las transacciones se bloquean y quedan en espera cuando piden bloquear un dato y éste ya está con un bloqueo (indiferente que sea compartido o exclusivo).
  + Por eso vemos que la instrucción 2 bloquea la transacción (T2) y la pone en espera, mismo con la instrucción 10 (T1).
  + Para el primer bloqueo que ocurre con T2, vemos que se ejecutan las instrucciones 3,4,5,6, donde la 6 libera el dato y ocurre una 6’ que es básicamente el lock compartido que ocurre en la instrucción 2 (para seguir con la instrucción 7).
  + Para el segundo bloqueo que ocurre con T1, vemos que T2 hace un lock compartido de B, y en la instrucción 10 T1 intenta bloquear el dato pero es puesto en espera. Ocurren las instrucciones 11-12 y existe un 12’ que aplica el bloqueo que quiso imponer la instrucción 10, para así seguir con la instrucción 13.
  + Informar (A+B) informa algo incorrecto, ya que informa 2950…
    - Se genera una inconsistencia de la información.
    - En resumen, el método del bloqueo aún puede generar inconsistencias.
    - Falta una vuelta de tuerca más.
* La solución es llevar los bloqueos de las transacciones al comienzo.

*Deadlock*

* Situación en la que una transacción espera un recurso de otra transacción y viceversa.
  + T1 pide el dato A, se lo dan.
  + T2 pide el dato B, se lo dan.
  + T1 pide el dato B, no se lo pueden dar xq T2 lo está acaparando.
  + T2 pide el dato A, no se lo pueden dar xq T1 lo está acaparando.
  + No se libera nada xq T1 espera de T2 y viceversa.
* Es el efecto secundario que ocurre de bloquear la información al principio, de usarla y liberarla.
* *Conclusiones*
  + Si los datos son liberados pronto 🡪 se evitan posibles deadlock.
  + Si los datos se mantienen bloqueados 🡪 se evitan inconsistencias.
  + Tengo que lidiar con deadlock o inconsistencia, cuál prefiero?
    - Se prefiere lidiar con el deadlock ya que el mismo no genera inconsistencia sobre la BD.
    - ¿Cómo soluciono un deadlock?
      * La solución es elegir a una víctima, elijo a una de las dos transacciones y la ABORTO.
      * La otra transacción puede empezar con sus instrucciones.

*Protocolos de bloqueo*

* Dos fases
  + Requiere que las transacciones hagan bloqueos en dos fases para evitar inconsistencias:
    - Fase de crecimiento: se obtienen datos.
    - Fase de decrecimiento: se liberan los datos.
    - Pido todo al principio, uso y libero al final.
  + Garantiza seriabilidad en conflictos, pero no evita situaciones de deadlock.
  + Como se consideran las operaciones
    - Fase de crecimiento: se piden bloqueos en orden: compartido, exclusivo.
    - Fase de decrecimiento: se liberan datos o se pasa de exclusivo a compartido.

*Protocolo basado en hora de entrada*

* El orden de ejecución es determinado por adelantado, no depende de quien llega primero.
* Cada transacción recibe una HDE (Hora de entrada)
  + Hora del servidor
  + o
  + Un contador
* Sí HDE(Ti) < HDE(Tj), Ti es anterior a Tj
* Cada dato
  + Hora en que se ejecutó el último WRITE.
  + Hora en que se ejecutó el último READ.
  + Las operaciones READ y WRITE que pueden entrar en conflicto se ejecutan y eventualmente fallan por HDE.
    - ¿Qué significa?
* Algoritmo de ejecución
  + Ti solicita READ(Q)
    - HDE(Ti) < HW(Q): se rechaza ya que Ti solicita un dato que fue escrito por una transacción posterior, estaría queriendo leer un dato que ya fue reemplazado.
    - HDE(Ti) >= HW(Q): ejecuta por que mi transacción quiere leer un dato que fue modificado antes que mi hora de entrada. Se establece HR(Q) = Max{HDE(Ti), HR(Ti)} (saca el máximo entre esos dos datos).
  + Ti solicita WRITE(Q)
    - HDE(Ti) < HR(Q): rechaza ya que Q es utilizado por otra transacción anteriormente y ésta supuso que Q no cambiaba.
    - HDE(Ti) < HW(q): rechaza por que Ti intenta escribir un valor viejo y obsoleto.
    - HDE(Ti) > [HW(Q) y HR(Q)]: ejecuta y HW(Q) se establece con HDE(Ti)
  + Sí Ti falla, y se rechaza entonces puede recomenzar con una nueva hora de entrada.

*Casos de concurrencia. Granularidad*

* A registros caso más normal de bloqueo.
* Otros casos de bloqueo
  + BD completa (quien puede querer bloquear la BD completa?)
  + Áreas (y un conjunto de áreas?)
  + Tablas (y un conjunto de tablas?)
  + Algunos DBMS permiten un bloqueo de atributos:
    - Se llama nivel de granularidad.
    - El nivel de granularidad de bloqueo depende de cada DBMS.

*Otras operaciones conflictivas*

* Delete(Q) requiere un uso completo del registro, ya que borra una tupla de la tabla.
* Insert(Q) el dato permanece bloqueado hasta que la operación finalice.
  + ¿Por qué bloquearía un dato que todavía nadie puede leer?
    - Puede que necesite insertar en más de una tabla a la vez.
      * Ejemplo: insertar una factura en una tabla facturas.
        + A su vez se deben insertar los renglones de dicha factura en la tabla renglones.

**Registro histórico en entornos concurrentes**

*Consideraciones del protocolo basado en bitácora*

* Existe un único buffer de datos compartidos y uno para la bitácora.
* Cada transacción tiene un área donde lleva sus datos.
* El retroceso de una transacción puede llevar al retroceso de otras transacciones.

*Retroceso en cascada*

* Falla una transacción 🡪 puede llevar a abortar otras.
  + Se abortan otras cuando hay datos en común usados en varias transacciones.
  + Se ejecuta T1, se ejecuta T2 y se ejecuta T3.
  + T1 falla, como T2 usa datos de T1 y T3 también. Entonces se abortan las 3.
* Puede llevar a deshacer una gran cantidad de trabajo.
  + Me la tengo que bancar.
  + Puede ocurrir pero no es muy habitual.

*Durabilidad*

* Puede ocurrir que Ti falle, y que Tj deba retrocederse, pero que Tj ya terminó.
* Transacción que alcance el estado de cometido, NO PUEDE RETROCEDER.
* Como debería actuar para cumplir la condición de durabilidad en Tj?.
  + Protocolo de bloqueo en dos fases: los bloqueos exclusivos deben conservarse hasta que Ti termine (cuando Ti alcance el estado de cometido libera los bloqueos exclusivos).
  + HDE, agrega un bit, para escribir el dato, además de lo analizado, revisar el bit: sí está en 0 procedemos, si está en 1 la transacción anterior no terminó, debo esperar.
  + Esto implica que las transacciones que conflictúan se hagan en SERIE, de forma necesaria.
    - No ocurre habitualmente.
  + También evito el retroceso en cascada.

*Bitácora*

* Es similar a los sistemas monousuario.
* Como proceder con checkpoints
  + Colocarlo cuando ninguna transacción esté activa. Puede que no exista el momento.
  + Checkpoint <L> L es una lista de transacciones activas al momento del checkpoint.
  + Básicamente: de acá para atrás está todo bien, pero revisá la lista de transacciones activas.
* Ante un fallo
  + UNDO y REDO según el caso.
  + Debemos buscar antes del checkpoint solo aquellas transacciones que estén en la lista.