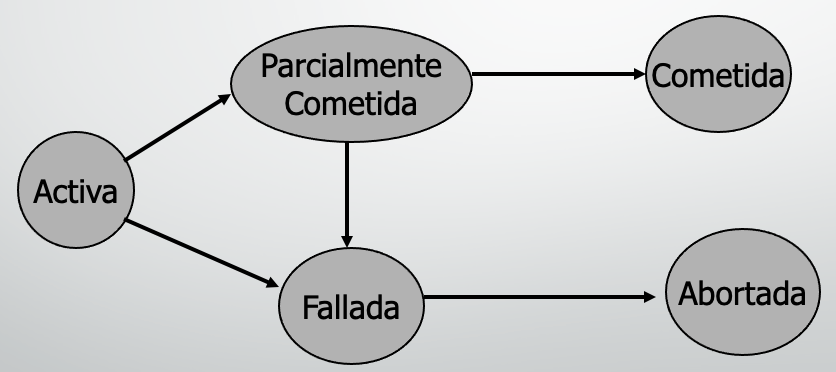
**Seguridad e Integridad de Datos**

Seguridad => Uso indebido de la base de datos y, eventualmente, rotura de la base de datos a partir de la mala intención.

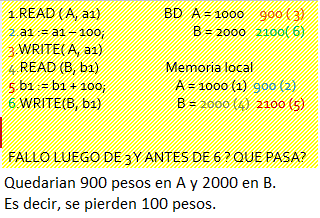
Integridad de Datos => Rotura de la información a partir de su uso cotidiano, y sin intención de que se produzca dicha rotura.

**Conceptos**

* Transacción: Colección de operaciones que forman una única unidad lógica de trabajo.
  + Propiedades ACID:
    - **Atomicidad**: Todas las operaciones de la transacción se ejecutan o no lo hace ninguna de ellas. (MONOUSUARIO)
    - **Consistencia**: La ejecución aislada de la transacción conserva la consistencia de la BD.
    - **Aislamiento** (isolation): Cada transacción ignora el resto de las transacciones que se ejecutan concurrentemente en el sistema, actúa cada una como única.
    - **Durabilidad**: Una transacción terminada con éxito realiza cambios permanentes en la BD, incluso si hay fallos en el sistema
  + Estados:
    - Activa: Estado inicial, estado normal durante la ejecución.
    - Parcialmente Cometida: Después de ejecutarse la última instrucción.
    - Fallada: Luego de descubrir que no puede seguir la ejecución normal.
    - Abortada: Después de haber retrocedido la transacción y restablecido la BD al estado anterior al comienzo de la transacción.
    - Cometida: Tras completarse con éxito.



* + ¿Por qué puede fallar en estado de parcialmente cometida?
    - Puede ocurrir por una escritura sobre la BD (Update).
    - ¿Por qué?, simple, las escrituras se hacen primero sobre Buffer, recordemos que es memoria RAM (volátil).
    - Puede ocurrir que se pierda dicha información, por lo que se puede llevar a un error que implique el paso al estado de transacción fallida.
  + Uso:
    - Las transacciones pueden generar problemas, sí, lo sabemos, y en base al tipo del sistema que tengamos, vamos a tener que actuar de una forma u otra.
      * Sistemas monousuario: en el sistema monousuario siempre hay una transacción ocurriendo en cada momento, al ser monousuario no se permiten varias transacciones en el mismo tiempo, por lo que de las propiedades ACID se eliminan la de consistencia, aislamiento y durabilidad, sólo debemos asegurar que se cumpla la propiedad de atomicidad.
      * Sistemas concurrentes.
      * Sistemas distribuidos.
      * Distribuidos (no se ve en ésta materia) y concurrentes no nos interesan por ahora.



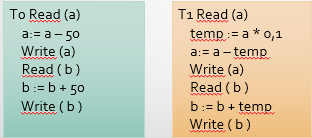
* + ¿Qué hacer luego de un fallo? QUIERO MIS 100 PESOS
    - Re-ejecutar la transacción fallada 🡪 NO SIRVE
    - Dejar el estado de la BD como está 🡪 NO SIRVE
  + Problema 🡪 Modificar la BD sin seguridad que la transacción se va a cometer.
    - Solución => Indicar las modificaciones (indicar qué voy a hacer antes de hacerlo).

Soluciones:

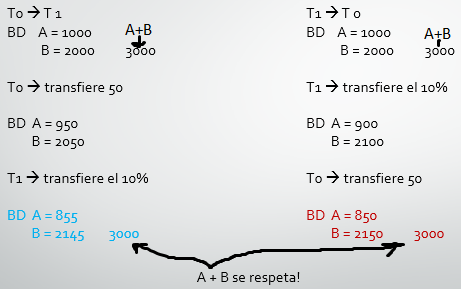
* **Registro Histórico (Bitácora o Log)**
  + Secuencia de actividades realizadas sobre la BD
  + Contenido de la bitácora **en orden** (es escrita antes de que se realice la transacción, estará en disco rígido y especifican las operaciones a futuro que se realizarán en la BD, es un archivo más)
    - <T iniciada> **T es transacción**
    - <T, E, Va, Vn>
      * Identificador de la transacción (T).
      * Identificador del elemento de datos (E, registra que valor se va a cambiar).
      * Valor anterior (Va, de ese atributo).
      * Valor nuevo (Vn, de ese atributo).
    - <T Commit> **Se agrega al final cuando la transacción fue exitosa.**
    - <T Abort> **Se agrega al final cuando la transacción fue fallida.**
  + Las operaciones sobre la BD deben almacenarse luego de guardar en disco el contenido de la Bitácora.
  + Técnicas:
    - Modificación diferida de la BD
      * Las operaciones write se aplazan hasta que la transacción esté parcialmente cometida (hasta el final), en ese momento se actualiza la bitácora y la BD.Recién con T parcialmente cometida, entonces se actualiza la BD.
      * Ante un fallo, y luego de recuperarse:
        + REDO (Ti), para todo Ti que tenga un Start y un Commit en la Bitácora.
        + Si no tiene Commit se ignora, dado que no llegó a hacer algo en la BD.
    - Modificación inmediata de la BD
      * La actualización de la BD se realiza mientras la transacción está activa y se va ejecutando.
      * Se necesita el valor viejo, pues los cambios se fueron efectuando.
      * Es un protocolo más complejo que modificación diferida, porque tiene otra operación adicional.
        + Tiene mejor distribuida la carga de trabajo (ya que diferida usa el disco rígido mucho al final de todo).
      * Ante un fallo, y luego recuperarse:
        + REDO (Ti), para todo Ti que tenga un Start y un Commit en la Bitácora.
        + UNDO (Ti), para todo Ti que tenga un Star y no un Commit.
* Transacción
  + - Condición de idempotencia: Puedo rehacer o deshacer una transacción de bitácora las veces que quiera y el resultado siempre es el mismo.
* Buffers de Bitácora
  + - Grabar en disco cada registro de bitácora insume gran costo de tiempo 🡪 Se utilizan buffers
      * Transacción está parcialmente cometida después de grabar en memoria no volátil el Commit en la Bitácora.
      * Un Commit en la bitácora en memoria no volátil, implica que todos los registros anteriores de esa transacción ya están en memoria no volátil.
      * SIEMPRE graba primero la Bitácora y luego la BD.
        + WRITE(Bitacora)
        + WRITE(BD)
        + OUTPUT(Bitacora)
        + OUTPUT(BD)
* Puntos de Verificación:
  + Ante un fallo, ¿Qué hacer?
    - REDO, UNDO: según el caso.
  + Revisar la bitácora:
    - ¿Desde el comienzo?: probablemente gran porcentaje del trabajo esté correcto y terminado, por eso lleva MUCHO tiempo la revisión de bitácora.
  + Checkpoints (solo para sistemas monousuario)
    - Se agregan periódicamente indicando desde allí hacia atrás todo OK (ósea, no se debe revisar nada en ese lugar).
    - Periodicidad en la cual se debe poner un checkpoint?, no hay respuesta
      * Checkpoints muy cerca entre sí: debo asegurar que el buffer debe estar bajado a disco, puede agregar mayor carga al sistema.
      * Checkpoints más lejos entre sí: en un caso de fallo debo revisar mucho.
      * Se debe buscar un punto intermedio.
* **Doble Paginación**
  + Paginación en la sombra
    - Ventaja: Menos accesos a disco
    - Desventaja: Complicada en un ambiente concurrente/distribuido
    - N páginas equivalente a páginas del SO
      * Tabla de páginas actual
      * Tabla de páginas sombra
      * Ambas tienen la misma información al comenzar
  + Ejecución de la operación escribir:
    - Ejecutar entrada(X) si página i-ésima no está todavía en memoria principal.
    - Si es la primer escritura sobre la página, modificar la tabla actual de páginas:
      * Encontrar una página en el disco NO utilizada
      * Indicar que a partir de ahora está ocupada
      * Modificar la tabla actual de página indicando que la i-ésima entrada ahora apunta a la nueva página
  + En caso de fallo y luego de la recuperación
    - Copia la tabla de páginas sombra en MP
    - Abort automáticos, se tiene la dirección de la página anterior sin las modificaciones
    - Ventajas:
      * Elimina la sobrecarga de escrituras de log
        + Recuperación más rápida (no existe REDO o UNDO)
    - Desventajas:
      * Sobrecarga en el compromiso: La técnica de paginación es por cada transacción.
      * Fragmentación de datos: Cambia la ubicación de los datos continuamente.
      * Garbage Collector: Ante un fallo queda una página que no es más referenciada.

Entornos Concurrentes:

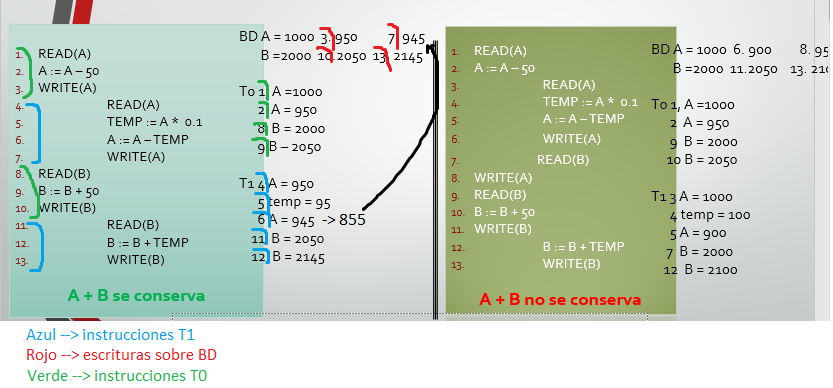
* Entorno Centralizado:
  + Varias transacciones ejecutándose simultáneamente compartiendo recursos.
  + Deben evitarse los mismos problemas de consistencia de datos.
  + Transacciones correctas, en ambientes concurrentes pueden llevar a fallos.
* Seriabilidad 🡪 Garantiza la consistencia de la BD



* 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
  + 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.
* OBSERVACIONES:

1. El programa debe conservar la consistencia
2. 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.
3. Una planificación concurrente debe equivaler a una planificación en serie
4. Sobre las instrucciones READ y WRITE son importantes y deben considerarse
   * + - Las operaciones ajenas a esas dos instrucciones son operaciones locales y no nos interesan para nuestro análisis.

* Conflicto en planificaciones serializables
  + I1, I2 instrucciones de T1 y T2
    - 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.

Métodos de Control de 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.

1. Bloqueo:
   * Compartido Lock\_c (dato) 🡪 solo lectura
   * Exclusivo Lock\_e (dato) 🡪 lectura/escritura
   * Las transacciones piden lo que necesitan.
   * Los bloqueos pueden ser compatibles y existir simultáneamente (compartidos)

* Una transacción debe:
  + Obtener el dato (si está libre, o compartido y solicita compartido)
  + Espera (otro caso)
  + Usar el dato
  + Liberarlo
* Se deben llevar los bloqueos de las transacciones al comienzo.
* Deadlock 🡪 Situación en la que una transacción espera un recurso de otra y viceversa.
  + Es el efecto secundario que ocurre de bloquear la información al principio, de usarla y liberarla.
    - 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.
* Conclusiones:
  + Si los datos se liberan pronto 🡪 Se evitan posibles deadlock.
  + Si los datos se mantienen bloqueados 🡪 Se evita inconsistencia (Mejor opción).
    - ¿Cómo soluciono situaciones de deadlock? 🡪 Elijo a la víctima, elijo a una de las dos transacciones y la aborto (deshago todo lo que hizo dicha transacción y libero los datos).
* Protocolos de bloqueo 🡪 Dos fases:
  + Requiere que las transacciones hagan bloqueos en dos fases:
    - 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 operaciones
    - Fase crecimiento: se piden bloqueos en orden: compartido, exclusivo
    - Fase decrecimiento: se liberan datos o se pasa de exclusivo a compartido

1. Basado en hora de entrada:
   * El orden de ejecución se determina por adelantado, no depende de quien llega primero
   * Cada transacción recibe una Hora De Entrada (HDE)
     + Hora del servidor
     + Un contador
   * Si HDE(Ti) < HDE(Tj), Ti es anterior
   * Cada dato registra:
     + 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.

* Algoritmo de ejecución:
  + Ti solicita READ(Q)
    - HDE(Ti) < HW(Q): rechazo (solicita un dato que fue escrito por una transacción)
    - HDE(Ti) >= HW(Q): ejecuta y se establece HR(Q)=Max{HDE(Ti), HR(Ti)}
  + Ti solicita WRITE(Q)
    - HDE(Ti) < HR(Q): rechazo (Q fue utilizado por otra transacción anteriormente y supuso que no cambiaba)
    - HDE(Ti) < HW(Q): rechazo (se intenta escribir un valor viejo, obsoleto)
    - HDE(Ti) > [HW(Q) y HR(Q)]: ejecuta y HW(Q) se establece con HDE(Ti)
  + Si Ti falla, y se rechaza entonces puede recomenzar con una nueva hora de entrada.

Casos de Concurrencia 🡪 Granularidad = nivel de bloqueo

* A registro 🡪 caso más normal
* Otros casos
  + BD completa
  + Áreas
  + Tablas

Otras operaciones conflictivas

* Delete(Q) requiere un uso completo del registro.
* Insert(Q) el dato permanece bloqueado hasta que la operación finalice.

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 🡪 Pueden llevar a abortar otras
  + Puede llevar a deshacer gran cantidad de trabajo.
* Durabilidad:
  + Puede ocurrir que falle Ti, y que Tj deba retrocederse, pero que Tj ya terminó. Transacción que alcance el estado de cometido NO puede retroceder.
  + Protocolo de bloqueo de dos fases: Los bloqueos exclusivos deben conservarse hasta que Ti termine (estado de cometido).
  + HDE, agrega un bit, para escribir el dato, además de lo analizado, revisar el bit si está en 0 proceder, si está en 1 la transacción anterior no terminó, esperar…
* Bitácora:
  + Similar sistemas monousuarios
  + Como proceder con checkpoint
    - Colocarlo cuando ninguna transacción esté activa. Puede que no exista el momento.
    - Checkpoint<L> L lista de transacciones activas al momento del checkpoint.
  + Ante un fallo
    - UNDO y REDO según el caso.
    - Debemos buscar antes del Checkpoint solo aquellas transacciones que estén en la lista.