# Tema 3.3: Recuperación

# Tema 5.3: Recuperación

- Clasificación de Fallos
- Estructura del Almacenamiento
- Recuperación y Atomicidad
- Recuperación basada en Log

# Clasificación de Fallos

- Fallo de Transacción:
  - Errores Lógicos: La transacción no se puede completar debido a alguna condición de error interna
  - Errores de Sistema: El sistema de base de datos debe terminar una transacción activa debido a una condición de error (ej: deadlock)
- Caída del Sistema: Un fallo de alimentación u otro fallo SW o HW que cause que el sistema se caiga
  - Supuesto de Fail-stop: El contenido del almacenamiento no-volátil se supone que no se corrompe por una caída del sistema
    - Los sistemas de base de datos tienen numerosas comprobaciones de integridad para prevenir corrupciones de datos en los discos
- Fallo de Disco: Una caída de una cabeza o un fallo similar de disco destruye todo o parte del almacenamiento de disco
  - La destrucción se supone que es detectable: los drivers de disco disponen de checksums para detectar fallos

# Algoritmos de Recuperación

- Los Algoritmos de Recuperación son técnicas para asegurar:
  - La consistencia de una base de datos, y
  - La atomicidad y durabilidad de una transacción

  a pesar de que existan fallos
- Los Algoritmos de Recuperación tienen dos partes
  1. Acciones tomadas **durante** el procesamiento normal de una transacción, para asegurar que existe suficiente información para recuperarse de fallos
  2. Acciones tomadas **tras un fallo**, para recuperar el contenido de una base de datos hasta un estado que asegure la atomicidad, la consistencia y la durabilidad

# Estructura del Almacenamiento

- Almacenamiento volátil:
  - NO sobrevive a caídas del sistema
  - Ejemplos: memoria principal, memoria caché
- Almacenamiento no-volátil:
  - Sobrevive a caídas del sistema
  - Ejemplos: disco, cinta, memoria flash, RAM no-volátil (alimentada por batería)
- Almacenamiento estable:
  - Una forma mítica de almacenamiento que sobrevive a todos los fallos
  - Aproximación: mantener múltiples copias en diferentes medios no-volátiles

# Acceso a Datos

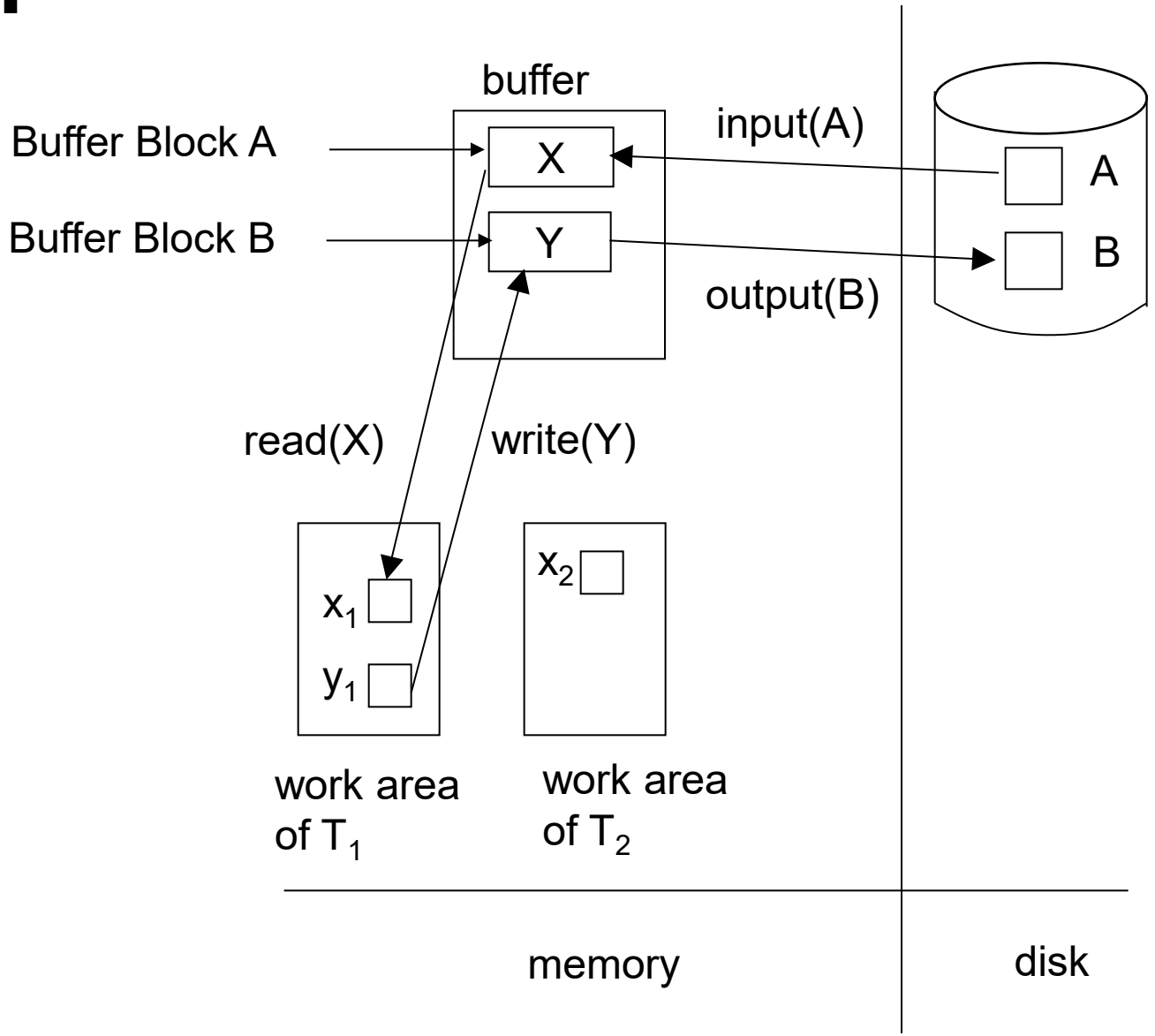
- **Bloques Físicos** son aquellos bloques que residen en disco
- **Bloques de Buffer** son aquellos bloques que residen temporalmente en memoria principal
- Los movimientos de bloques entre disco y memoria principal se inician mediante las siguientes dos operaciones:
  - **input**(*B*) transfiere el bloque físico *B* a memoria principal
  - **output**(*B*) transfiere el bloque de buffer *B* a disco, y sustituye el bloque físico correspondiente allí
- Cada transacción $T_i$ tiene su área de trabajo privada en la que se guardan copias locales de todos los ítems de datos y a los que se acceden y se actualizan por ella
  - La copia local de $T_i$ de un ítem de datos *X* se llama $x_i$
- Supondremos, por simplicidad, que cada ítem de datos se almacena y cabe en un solo bloque

# Acceso a Datos (cont.)

- Una transacción transfiere ítems de datos entre bloques de buffer de sistema y su área de trabajo privada usando las siguientes operaciones:
  - **read**($X$) asigna el valor del ítem de datos $X$ a la variable local $x_i$
  - **write**($X$) asigna el valor de la variable local $x_i$ al ítem de datos {$X$} en el bloque de buffer
  - Ambas operaciones pueden necesitar ejecutar la instrucción **input**($B_X$) antes de la asignación, si el bloque $B_X$ en el que reside $X$ no está ya en memoria

- **output**($B_X$) no necesita realizarse a continuación de **write**($X$). El sistema puede realizar la operación de **output** cuando le venga bien

# Ejemplo de Acceso a Datos



# Recuperación y Atomicidad

- Modificar la base de datos sin asegurar que la transacción se comprometerá puede dejar la base de datos en un estado inconsistente
- Consideremos la transacción Ti que traspasa 50 € de la cuenta A a la cuenta B; el objetivo es o bien realizar todas las modificaciones en la base de datos efectuadas por Ti o bien ninguna
- Ti puede requerir varias operaciones de output (para volcar A y B). Un fallo puede ocurrir tras efectuar una de estas modificaciones pero antes de que todas ellas se hayan realizado

# Recuperación y Atomicidad (cont.)

- Para asegurar la atomicidad a pesar de fallos, primero se escribirá información describiendo las modificaciones en almacenamiento estable sin modificar la base de datos misma
- Estudiaremos dos aproximaciones:
  - **Recuperación basada en Log**, y
  - **Recuperación basada en Shadow-paging**
- Supondremos (inicialmente) que las transacciones se ejecutan secuencialmente, esto es, una tras de otra

# Recuperación basada en Log

- Un **log** se mantiene en almacenamiento estable
  - El **log** es una secuencia de **registros de log**, y mantiene un histórico de las actividades de actualización sobre la base de datos
- Cuando la transacción $T_i$ comienza, ella misma se registra escribiendo un registro <$T_i$ **start**> en el log
- *Antes* que $T_i$ ejecute **write**($X$), un registro <$T_i$, $X$, $V_1$, $V_2$> se escribe en el log, donde $V_1$ es el valor de $X$ antes del write, y $V_2$ es el valor a escribir en $X$
  - El registro del log anota que: $T_i$ ha realizado write en el item de datos $X$, $X$ tenía el valor $V_1$ antes del write, y tendrá el valor $V_2$ tras el write
- Cuando $T_i$ termine su última instrucción, se escribirá un registro <$T_i$ **commi**t> en el log
- Supondremos que los registros de log se escriben directamente en almacenamiento estable (esto es, no existe buffer intermedios)
- Dos aproximaciones
  - Modificación Diferida de BB.DD.
  - Modificación Inmediata de BB.DD.

# Modificación Diferida de BB.DD.

- El esquema de modificación diferida de **base de datos** registra todas las modificaciones en el log, pero pospone todos los **writes** hasta tras realizar un commit parcial
- Supone que las transacciones se ejecutan secuencialmente
- Las transacciones empiezan escribiendo un registro <$T_i$ ***start***> en el log
- Una operación **write**(*X*), hace que se escriba un registro <$T_i$, *X*, *V*> en el log, donde *V* es el nuevo valor de *X*
  - En este esquema no es necesario el valor antiguo
- No se ejecuta la escritura de *X* en este momento, sino que se pospone
- Cuando $T_i$ se consolida parcialmente, se escribe <$T_i$ **commit**> en el log
- Finalmente, los registros de log se leen y se usan para ejecutar realmente los writes diferidos previamente

# Modificación Diferida de BB.DD. (cont.)

- Durante la recuperación tras una caída, se necesita rehacer (redo) una transacción si y solo si ambos <$T_i$ **start**> y <$T_i$ **commit**> se encuentran en el log
- Rehacer una transacción $T_i$ (**redo** $T_i$) pone los valores de los ítems de datos actualizados por la transacción, en los nuevos valores
- Las caídas pueden ocurrir mientras
  - la transacción esta ejecutando las actualizaciones originales, o
    - Transacciones ejemplo $T_0$ y $T_1$ ($T_0$ se ejecuta antes que $T_1$):
  - Mientras se esta llevando a cabo la recuperación

| | | | |
|---|---|---|---|
| $T_0$: | **read** ($A$) | $T_1$: | **read** ($C$) |
| | $A := A - 50$ | | $C := C - 100$ |
| | **write** ($A$) | | **write** ($C$) |
| | **read** ($B$) | | |
| | $B := B + 50$ | | |
| | **write** ($B$) | | |

# Modificación Diferida de BB.DD. (cont.)

- Estado del log según aparece en tres instantes de tiempo

| (a) | (b) | (c) |
|---|---|---|
| <$T_0$ start> | <$T_0$ start> | <$T_0$ start> |
| <$T_0$, $A$, 950> | <$T_0$, $A$, 950> | <$T_0$, $A$, 950> |
| <$T_0$, $B$, 2050> | <$T_0$, $B$, 2050> | <$T_0$, $B$, 2050> |
| | <$T_0$ commit> | <$T_0$ commit> |
| | <$T_1$ start> | <$T_1$ start> |
| | <$T_1$, $C$, 600> | <$T_1$, $C$, 600> |
| | | <$T_1$ commit> |

- Si el log en almacenamiento estable en el momento de la caída fuera, según el caso,...
  - (a) No se necesita realizar acciones de **redo**
  - (b) Se debe realizar **redo**($T_0$) puesto que <$T_0$ **commit**> está presente
  - (c) Se debe realizar **redo**($T_0$) seguido de **redo**($T_1$) puesto que <$T_0$ **commit**> y <$T_i$ commit> están presentes

# Modificación Inmediata de BB.DD. (cont.)

- El esquema de **modificación inmediata** permite a una base de datos que las modificaciones de una transacción no consolidada se realicen según se ejecuten las operaciones de **write**
  - Puesto que se pueden necesitar la operación de **undo**, los registros del log deben incluir tanto el valor viejo como el nuevo
- Los registros del log se deben escribir ***antes*** de que el ítem se escriba en la base de datos
  - Supondremos que el registro de log se escribe directamente en almacenamiento estable
  - Extension: posponer la escritura del registro de log, siempre que antes de ejecutar una operación **output**(*B*) para un bloque de datos B, todos los registros de log correspondientes al ítem *B* se hayan volcado físicamente (**flush**) a almacenamiento estable
- El volcado de bloques actualizados puede ocurrir en cualquier momento antes o después de consolidar (**commit**) la transacción
- El orden en el que los bloques se vuelcan (**output**) a disco puede ser diferente del orden en el que se escriben (**write**)

# Modificación Inmediata de BB.DD. Ejemplo

| Log | Write | Output |
|---|---|---|
| <$T_0$ **start**> | | |
| <$T_0$, A, 1000, 950> | | |
| <$T_0$, B, 2000, 2050> | | |
| | $A = 950$ | |
| | $B = 2050$ | |
| <$T_0$ **commit**> | | |
| <$T_1$ **start**> | | |
| <$T_1$, C, 700, 600> | | |
| | $C = 600$ | |
| | | $B_B$, $B_C$ |
| <$T_1$ **commit**> | | |
| | | $B_A$ |

- Nota: $B_X$ denota al bloque que contiene a $X$

# Modificación Inmediata de BB.DD. (cont.)

- El procedimiento de recuperación consta de dos operaciones:
  - **undo**($T_i$) recupera el valor de todos los ítems de datos actualizados por $T_i$ a sus valores originales, yendo **hacia atrás** desde el **ultimo** registro para $T_i$
  - **redo**($T_i$) coloca el valor de todos los ítems de datos actualizados por $T_i$ a los nuevos valores, yendo **hacia adelante** desde el **primer** registro para $T_i$
- Las dos operaciones deben ser idempotentes: Esto es, incluso si la operación se ejecuta múltiples veces el efecto es el mismo que si se ejecutara una vez
  - Necesario: las operaciones podrían re-ejecutarse durante la recuperación
- Recuperación tras un fallo:
  - Se necesita deshacer la transacción $T_i$ si el log contiene el registro <$T_i$ start>, pero NO contiene el registro <$T_i$ commit>
  - Se necesita rehacer la transacción $T_i$ si el log contiene AMBOS, el registro <$T_i$ start> Y el registro <$T_i$ commit>
  - Las operaciones de deshacer (**undo**) se ejecutan primero, después las operaciones de rehacer (**redo**)

# Modificación Inmediata de BB.DD. Ejemplo de Recuperación

- Estado del log según aparece en tres instantes de tiempo

| (a) | (b) | (c) |
|---|---|---|
| <$T_0$ start> | <$T_0$ start> | <$T_0$ start> |
| <$T_0$, $A$, 1000, 950> | <$T_0$, $A$, 1000, 950> | <$T_0$, $A$, 1000, 950> |
| <$T_0$, $B$, 2000, 2050> | <$T_0$, $B$, 2000, 2050> | <$T_0$, $B$, 2000, 2050> |
| | <$T_0$ commit> | <$T_0$ commit> |
| | <$T_1$ start> | <$T_1$ start> |
| | <$T_1$, $C$, 700, 600> | <$T_1$, $C$, 700, 600> |
| | | <$T_1$ commit> |

Las acciones de recuperación en cada caso de los anteriores son:

(a) undo ($T_0$): se restaura B a 2000 y A a 1000

(b) undo ($T_1$) y redo ($T_0$): se restaura C a 700, y después A y B toman los valores de 950 y 2050 respectivamente

(c) redo ($T_0$) y redo ($T_1$): A y B toman los valores de 950 y 2050 respectivamente. Luego C toma el valor 600

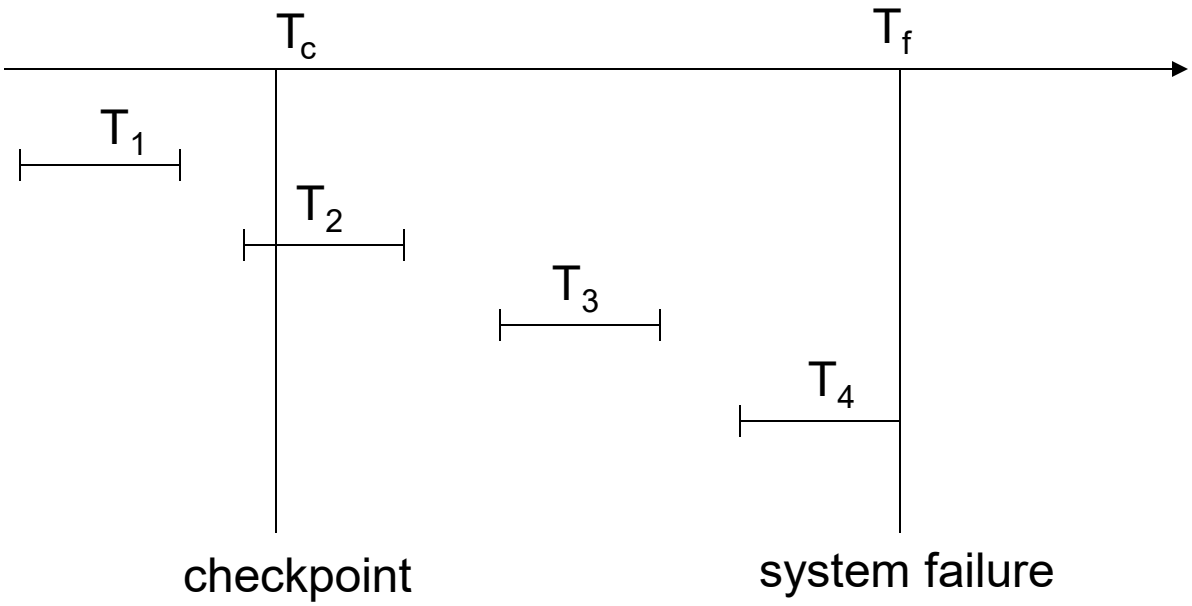
# Checkpoints

- Problemas en el procedimiento de recuperación (comentados anteriormente):
  1. Buscar en el log entero es caro en tiempo
  2. Se pueden rehacer innecesariamente transacciones que ya habían volcado sus actualizaciones en la base de datos
- Ajustar el procedimiento de recuperación al ejecutar periódicamente **checkpoints**
  1. Escribir un registro de log <**Start checkpoint>** en almacenamiento estable
  2. Volcar (**output**) todos los registros de log que residan en memoria principal en almacenamiento estable
  3. Volcar (**output**) todos los registros modificados de buffer en el disco
  4. Escribir un registro de log <**End checkpoint**> en almacenamiento estable, para indicar que el proceso ha finalizado con éxito.

# Checkpoints (cont.)

- Durante la recuperación necesitamos considerar **sólo** la transacción $T_i$ más reciente que empezara antes del checkpoint, y las transacciones que empezaron tras $T_i$
  1. Buscar hacia atrás desde el final del log, hasta encontrar el registro de log <**Start checkpoint**> más reciente
  2. Continuar buscando hacia atrás hasta encontrar un registro <$T_i$ **start**>
  3. Se necesita considerar la parte del log que sigue al registro **start**. Se puede ignorar durante la recuperación la parte anterior del log, y se puede borrar cuando se desee
  4. Para todas las transacciones (empezando por $T_i$ o posteriores) que no tengan <$T_i$ **commit**>, ejecutar **undo(***$T_i$***)** (Ejecutar solo en caso de modificación inmediata)
  5. Buscando hacia delante en el log, para todas las transacciones empezando desde $T_i$ o posteriores con un <$T_i$ **commit**>, ejecutar **redo(***$T_i$***)**

# Checkpoints. Ejemplo




- Se puede ignorar $T_1$ (las actualizaciones ya están volcadas a disco debido al *checkpoint*)
- Rehacer $T_2$ y $T_3$
- Deshacer $T_4$

# Recuperación con Transacciones Concurrentes

- Modificaremos el esquema de recuperación basado en log para permitir ejecución concurrente de múltiples transacciones
  - Todas las transacciones comparten un único buffer de disco y un único log
  - Un bloque de buffer puede tener ítems actualizados por una o más transacciones
- Supondremos control de concurrencia usando un bloqueo estricto de dos fases (i.e. Las actualizaciones de una transacción no consolidada no deberían ser visibles a otras transacciones)
  - De otra forma ¿cómo se podría deshacer si T1 modifica A, luego T2 modifica A y consolida, y finalmente T1 tiene que abortar?
- La gestión del Log se realiza como antes
  - Los registros de log de diferentes transacciones pueden estar entremezcladas en el log
- La técnica de checkpoint y las acciones que se toman durante la recuperación tienen que cambiarse
  - Puesto que varias transacciones pueden estar activas cuando se ejecuta un checkpoint

# Recuperación con Transacciones Concurrentes (cont.)

- Los checkpoints se realizan como antes, excepto que el registro de log del checkpoint log record ahora tiene la forma
  <**Start checkpoint** *L*>
  donde *L* es la lista de las transacciones activas en el momento del checkpoint
  - Supondremos que no hay actualizaciones pendientes mientras se ejecuta el checkpoint (aunque no lo necesitaremos mas adelante)
- Cuando el sistema se recupera de una caída, lo primero que hace es:
  1. Iniciar dos listas *undo-list* y *redo-list* a valores vacíos
  2. Buscar en el log hacia atrás desde el final, parándose cuando encuentra el primer registro <**Start checkpoint** *L*>
     Para cada registro encontrado durante esta búsqueda:
     - Si el registro es <$T_i$ **commit**>, añadir $T_i$ *a redo-list*
     - Si el registro es <$T_i$ **start**>, entonces si $T_i$ no está en *redo-list*, añadir $T_i$ a *undo-list*
  3. Para cada $T_i$ en *L*, si $T_i$ no está en *redo-list*, añadir $T_i$ a *undo-list*

# Recuperación con Transacciones Concurrentes (cont.)

- En este punto *undo-list* consta de transacciones incompletas que deben deshacerse, y *redo-list* consta de transacciones terminadas que deben rehacerse
- La recuperación continúa como sigue:
  1. Buscar hacia atrás en el log desde el registro más reciente, parándose cuando se hayan encontrado registros <$T_i$ **start**> para cada $T_i$ en *undo-list*
     - Durante la búsqueda, ejecutar **undo** para cada registro de log que pertenezca a las transacciones en *undo-list*
  2. Localizar el registro <**Start checkpoint** *L*> más reciente
  3. Buscar hacia delante en el log desde este registro <**Start checkpoint** *L*> hasta el final del log
     - Durante la búsqueda, ejecutar **redo** para cada registro de log que pertenezca a las transacciones en *redo-list*

# Recuperación. Ejemplo

- Seguir los pasos del algoritmo de recuperación en el siguiente log:

<$T_0$ **start**>
<$T_0$, *A*, 0, 10>
<$T_0$ **commit**>
<$T_1$ **start**> /* Busqueda del paso 1 llega hasta aquí */
<$T_1$, *B*, 0, 10>
<$T_2$ **start**>
<$T_2$, *C*, 0, 10>
<$T_2$, *C*, 10, 20>
<Start checkpoint {$T_1$, $T_2$}>
<End checkpoint >
<$T_3$ **start**>
<$T_3$, *A*, 10, 20>
<$T_3$, *D*, 0, 10>
<$T_3$ **commit**>

# Recuperación. Ejemplo

- En el caso anterior se suspende ejecución de transacciones en checkpoint, pero puede darse el caso de que no sea así.

<$T_0$ **start**>
<$T_0$, *A*, 0, 10>
<$T_0$ **commit**>
<$T_1$ **start**> /* Busqueda del paso 1 llega hasta aquí */
<$T_1$, *B*, 0, 10>
<$T_2$ **start**>
<$T_2$, *C*, 0, 10>
<Start checkpoint {$T_1$, $T_2$}>
<$T_2$, *C*, 10, 20>
<End checkpoint >
<$T_3$ **start**>
<$T_3$, *A*, 10, 20>
<$T_3$, *D*, 0, 10>
<$T_3$ **commit**>