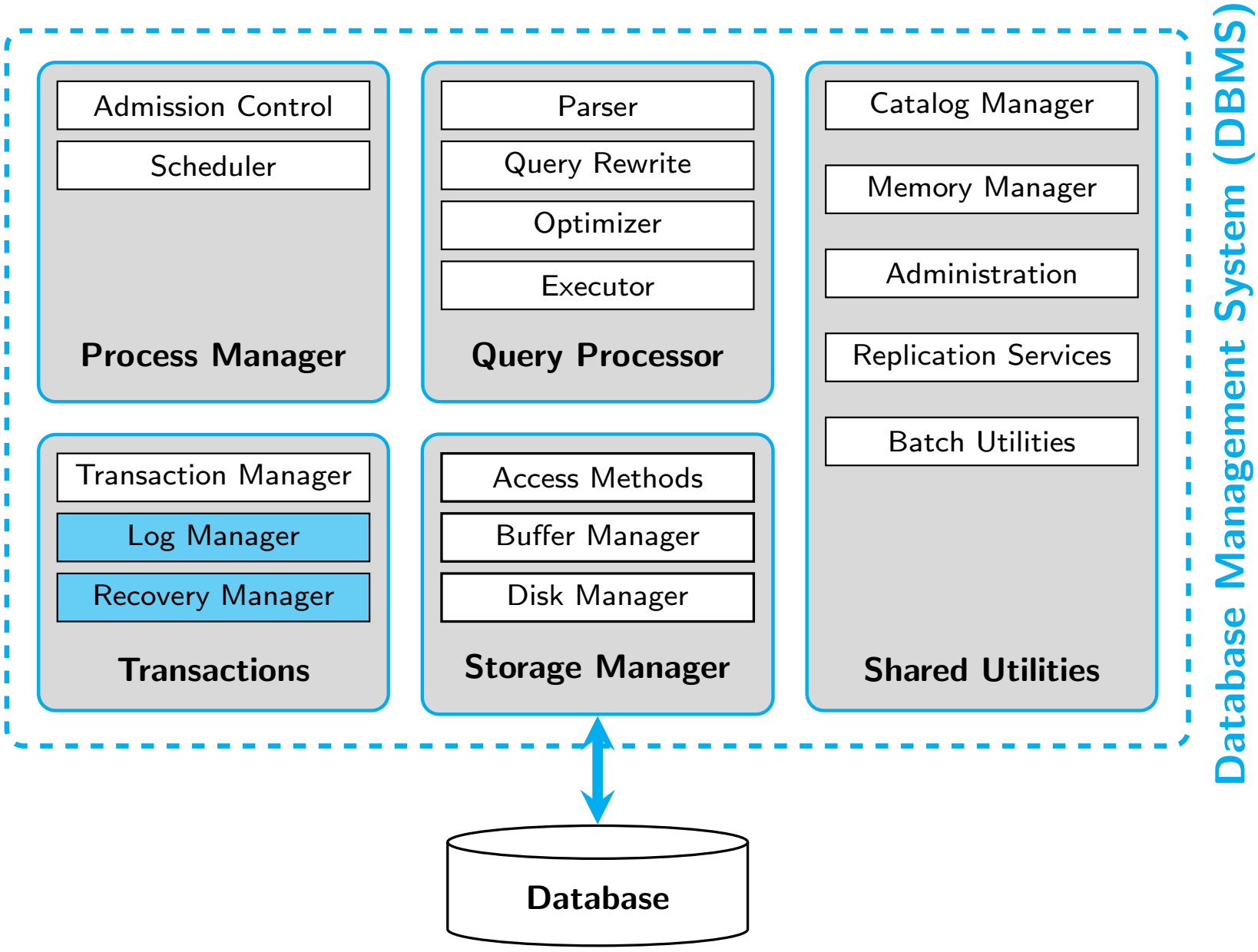
Undo-redo logging

Clase 21

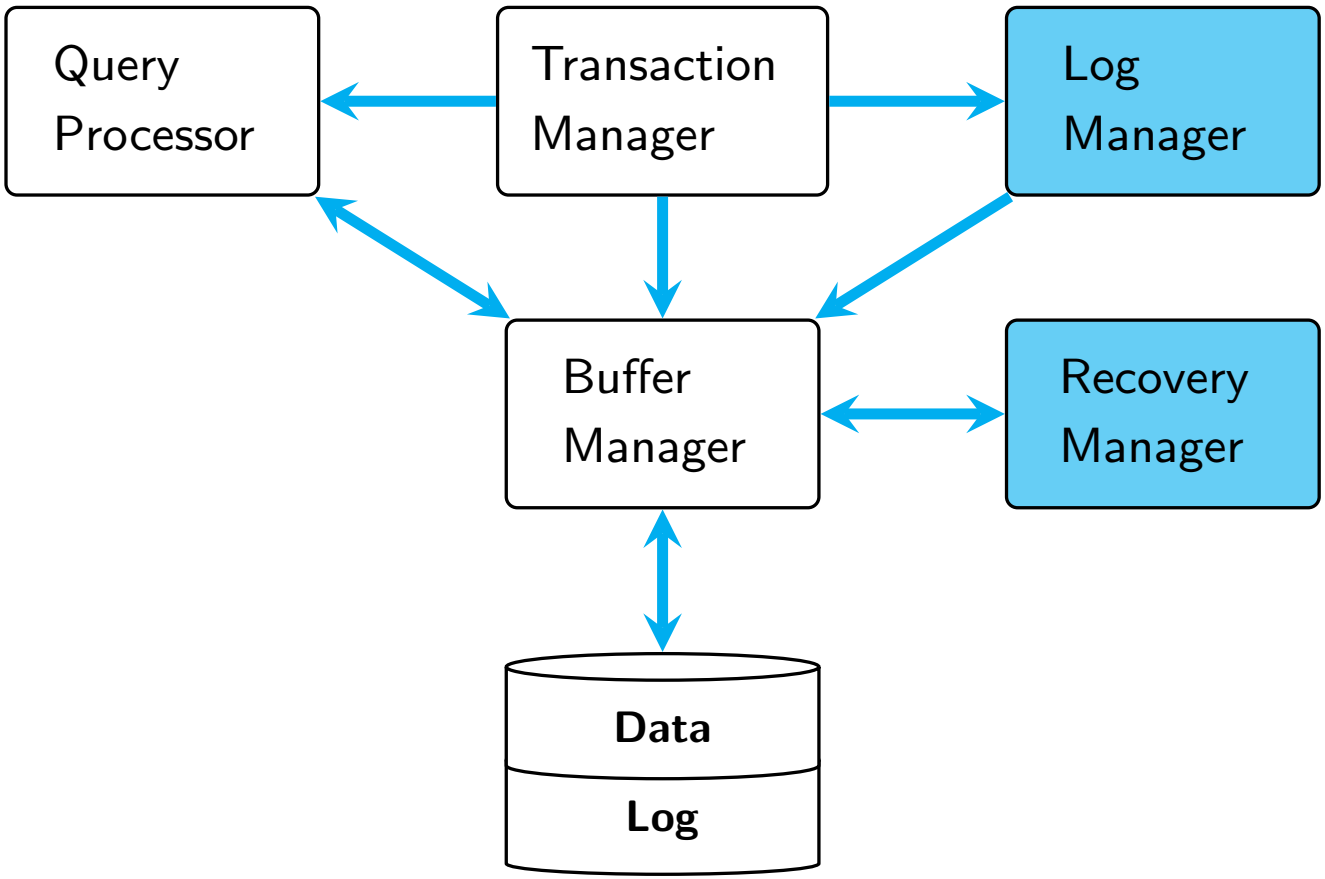
IIC 3413

Prof. Cristian Riveros

Recuperación de fallas en el sistema



Zoom a la arquitectura de las transacciones



Propiedades "acidas"

Atomicity: Se ejecuta todos los pasos de una transacción o no se ejecuta nada.

Consistency: Al terminar una transacción los datos deben estar en un estado consistente.

Isolation: Cada transacción se ejecuta sin ser interferida por otras transacciones.

Durability: Si un transacción hace commit, sus cambios sobreviviran cualquier tipo de falla.

Queremos asegurar **atomicity** y **durability**.

¿qué puede salir mal para **atomicity** o **durability**?

Ejemplo

T	t	Mem B_1	Mem B_2	B_1	B_2
PIN(B_1)		1000		1000	1000
READ(B_1, t)	1000				
$t := t - 100$	900				
WRITE(B_1, t)		900			
UNPIN(B_1)					
FLUSH(B_1)				900	
PIN(B_2)			1000		
READ(B_2, t)	1000				
$t := t + 100$	1100				
WRITE(B_2, t)			1100		
UNPIN(B_1)					
ERROR	✗	✗	✗	900	1000

Steal y force protocolos (recordatorio)

Steal frame:

¿Puede un elemento X en memoria ser escrito a disco antes de que la transacción T *termine?*

Force page:

¿Es necesario que una transacción haga `FLUSH` *de todos los elementos modificados inmediatamente antes o después de un* `COMMIT`*?*

- **NO-Steal** + **Force**: fácil de recuperar.
- **Steal** + **NO-Force**: mayor performance.

Queremos un **Steal** + **NO-Force** buffer manager!

Tres tipos de logging/recovery

1. *undo* - logging.
2. *redo* - logging.
3. *undo/redo* - logging.

Outline

Undo logging

Redo logging

Undo-redo logging

Concurrencia y recuperación

Outline

Undo logging

Log records:

- < `START` **T** >
- < `COMMIT` **T** >
- < `ABORT` **T** >
- < **T**, X , t >: donde t es el valor **antiguo** en X.

Undo-log registra el valor antiguo!

Reglas de undo-logging

Regla U_1:

"Si T *modifica* X*, el log record* $<$ T, X , $t>$ *debe ser escrito antes que* X *sea escrito en disco."*

Regla U_2:

"Si T *hace commit, el log record* $<$ `COMMIT` T $>$ *debe ser escrito justo después que todos los datos modificados por* T *esten almacenados en disco"*

Reglas de undo-logging

Reglas traducidas:

1. Escribir en el log $< \mathbf{T},\ X\ , t >$.
2. Escribir los datos a disco.
3. Escribir $<$ `COMMIT` $\mathbf{T} >$.

¿cómo nos aseguramos que
el log se escribe antes que la página en disco?

(usamos/definimos el método `FLUSH-LOG`)

Reglas de undo-logging

Ejemplo

T	t	Mem B_1	Mem B_2	B_1	B_2	LOG
						$<$ START **T** $>$
READ(B_1, t)	1000	1000		1000	1000	
$t := t - 100$	900					
WRITE(B_1, t)		900				$<$ **T**, B_1, 1000 $>$
READ(B_2, t)	1000		1000			
$t := t + 100$	1100					
WRITE(B_2, t)			1100			$<$ **T**, B_2, 1000 $>$
FLUSH-LOG						
FLUSH(B_1)				900		
FLUSH(B_2)					1100	
						$<$ COMMIT **T** $>$
FLUSH-LOG						

Recuperación de fallas con undo-logging

¿qué pudo andar mal?

Miremos el log:

Log en disco: . . . < START **T** > . . . < COMMIT **T** > . . . ✓

. . . < START **T** > . . . < ABORT **T** > . . . ✓

. . . < START **T** > . . . ✗

Recuperación de fallas con undo-logging

Procesamos el log desde el **final** hasta el **principio**:

1. $<$ `COMMIT` **T** $>$: marcar **T** como realizada.
2. $<$ `ABORT` **T** $>$: marcar **T** como realizada.
3. $<$ **T**, X, t $>$: restituir $X := t$ en disco si **T** no fue realizada.
4. $<$ `START` **T** $>$: ignorar.

Recuperación de fallas con undo-logging

Ejemplo

T	t	Mem B_1	Mem B_2	B_1	B_2	LOG
						< START **T** >
READ(B_1, t)	1000	1000		1000	1000	
$t := t - 100$	900					
WRITE(B_1, t)		900				< **T**, B_1, 1000 >
READ(B_2, t)	1000		1000			
$t := t + 100$	1100					
WRITE(B_2, t)			1100			< **T**, B_2, 1000 >
FLUSH-LOG						
FLUSH(B_1)				900		
FLUSH(B_2)					1100	
						< COMMIT **T** >
FLUSH-LOG						

¿qué pasa si el sistema falla en alguna instrucción?

Recuperación de fallas con undo-logging

Algunas preguntas sin resolver:

1. ¿Qué ocurre si el sistema falla otra vez en el momento del recovery?
2. ¿Hasta donde tenemos que leer el log?
3. ¿Es posible "truncar" el log en algún minuto?

Checkpoints

Checkpoint de la base de datos (periodicamente):

1. Dejamos de recibir nuevas transacciones.
2. Esperamos hasta que todas las transacciones actuales terminen.
3. `FLUSH` del log a disco.
4. Escribir < `CKPT` > al log y hacer `FLUSH-LOG`.
5. Reanudar las transacciones.

¿cómo usamos los checkpoints?

Checkpoints

Uso del log record < `CKPT` >:

- Hacemos undo recovery hasta que veamos < `CKPT` >.
- No es necesario seguir con undo recovery después de este punto.

¿Algún problema?

- Es casi necesario "**apagar**" el DBMS para hacer checkpoint.

¿cómo hacemos checkpoint sin apagar nuestro DBMS?

Nonquiescent checkpointing

Quiescent = ser tranquilo, quedarse quieto, o descansar.

Non-**quiescent** = lo contrario

(permitir nuevas transacciones)

Nonquiescent checkpointing

1. Escribimos un log record:

$$< \texttt{START CKPT}(\mathbf{T}_1, \ldots, \mathbf{T}_n) > .$$

   - $\mathbf{T}_1, \ldots, \mathbf{T}_n$ es la lista de transacciones activas.

2. Esperamos hasta que $\mathbf{T}_1, \ldots, \mathbf{T}_n$ terminen.
   - Sin restringir el comienzo de nuevas transacciones.

3. Cuando $\mathbf{T}_1, \ldots, \mathbf{T}_n$ hayan terminado escribimos el log record:

$$< \texttt{END CKPT} > .$$

¿cómo usamos el checkpoint para hacer el recovery?

Ejemplo (1)

Considere el siguiente undo-log después de una falla:

$$< \texttt{START}\ \mathbf{T}_1 >$$

$$< \mathbf{T}_1, A, 5 >$$

$$< \texttt{START}\ \mathbf{T}_2 >$$

$$< \mathbf{T}_2, B, 10 >$$

$$< \texttt{START CKPT}(\mathbf{T}_1, \mathbf{T}_2) >$$

$$< \mathbf{T}_2, C, 15 >$$

$$< \texttt{START}\ \mathbf{T}_3 >$$

$$< \mathbf{T}_1, D, 20 >$$

$$< \texttt{COMMIT}\ \mathbf{T}_1 >$$

$$< \mathbf{T}_3, E, 25 >$$

$$< \texttt{COMMIT}\ \mathbf{T}_2 >$$

$$< \texttt{END CKPT} >$$

¿cómo usamos el checkpoint para hacer el recovery?

Ejemplo (2)

Considere el siguiente undo-log después de una falla:

$$< \texttt{START}\ \mathbf{T}_1 >$$

$$< \mathbf{T}_1, A, 5 >$$

$$< \texttt{START}\ \mathbf{T}_2 >$$

$$< \mathbf{T}_2, B, 10 >$$

$$< \texttt{START CKPT}(\mathbf{T}_1, \mathbf{T}_2) >$$

$$< \mathbf{T}_2, C, 15 >$$

$$< \texttt{START}\ \mathbf{T}_3 >$$

$$< \mathbf{T}_1, D, 20 >$$

$$< \texttt{COMMIT}\ \mathbf{T}_1 >$$

$$< \mathbf{T}_3, E, 25 >$$

Implementación de ROLLBACK

¿cómo hacemos ROLLBACK?

Usamos **undo**-logo para implementar ROLLBACK:

- Optimización: usar referencia entre logs records.

LSN = Log Sequence Number

Outline

Redo logging

¿inconvenientes con **undo**-logging?

- No es posible hacer `COMMIT` antes de almacenar los datos en disco.
- Estamos obligados a usar una política FORCE pages.

Redo vs undo

	undo	**redo**
Trans. incompletas	cancelarlas	ignorarlas
Trans. commiteadas	ignorarlas	repetirlas
Escribir `COMMIT`	después de alm. en disco	antes de alm. en disco
Update log record	valores antiguos	valores nuevos

Log records para redo logging

Log records:

- < `START` **T** >
- < `COMMIT` **T** >
- < `ABORT` **T** >
- < **T**, X , t >: donde t es el valor **nuevo** en X.

Regla principal de redo-logging

Regla R_1:

> *"Antes de modificar cualquier elemento X en disco, es necesario que todos los logs records estén almacenados en disco, incluido el* `COMMIT` *log record."*

En otras palabras:

1. Escribir en el log $<$ **T**, X , t $>$.
2. Escribir $<$ `COMMIT` **T** $>$.
3. Escribir los datos a disco.

Esto es al revés de undo-logging!

Reglas de redo-logging

Ejemplo

T	t	Mem B_1	Mem B_2	B_1	B_2	LOG
						$<$ START **T** $>$
READ(B_1, t)	1000	1000		1000	1000	
$t := t - 100$	900					
WRITE(B_1, t)		900				$<$ **T**, B_1, 900 $>$
READ(B_2, t)	1000		1000			
$t := t + 100$	1100					
WRITE(B_2, t)			1100			$<$ **T**, B_2, 1100 $>$
						$<$ COMMIT **T** $>$
FLUSH-LOG						
UNPIN(B_1)				900		
UNPIN(B_2)					1100	

Recuperación de fallas con redo-logging

De nuevo miramos el log:

Log en disco: . . . < START **T** > . . . < COMMIT **T** > . . . ✗

. . . < START **T** > . . . < ABORT **T** > . . . ✓

. . . < START **T** > . . . ✓

Recuperación de fallas con redo-logging

1. Identificamos las transacciones que hicieron `COMMIT`.
2. Hacemos un scan desde el principio.
3. Para cada log record $< \mathbf{T}, X, v >$:
   - Si **T** no hizo `COMMIT`: no hacer nada.
   - Si **T** hizo `COMMIT`: reescribir con el valor v.
4. Para cada transacción incompleta **T**, escribir en el log $<$ `ABORT` **T** $>$.

Recuperación de fallas con redo-logging

Ejemplo

T	t	Mem B_1	Mem B_2	B_1	B_2	LOG
						$<$ START **T** $>$
READ(B_1, t)	1000	1000		1000	1000	
$t := t - 100$	900					
WRITE(B_1, t)		900				$<$ **T**, B_1, 900 $>$
READ(B_2, t)	1000		1000			
$t := t + 100$	1100					
WRITE(B_2, t)			1100			$<$ **T**, B_2, 1100 $>$
						$<$ COMMIT **T** $>$
FLUSH-LOG						
UNPIN(B_1)				900		
UNPIN(B_2)					1100	

¿qué ocurre si el sistema falla en alguna instrucción?

Checkpoints en redo-logging

- ¿Qué diferencia hay con undo-logging?
- ¿Qué debiéramos asegurar en un checkpoint?

Checkpoints en redo-logging

1. Escribir < `START CKPT`($\mathbf{T}_1, \ldots, \mathbf{T}_n$) > y hacer `FLUSH-LOG`.
   - $\mathbf{T}_1, \ldots, \mathbf{T}_n$ es la lista de transacciones activas y sin `COMMIT`.
2. Escribir a disco **solo** los **objetos** modificados por transacciones en estado `COMMIT`.
   - Sin restringir el comienzo de nuevas transacciones.
3. Escribir < `END CKPT` > y hacer `FLUSH-LOG`.

¿qué hacemos con las páginas
que comparten transacciones comiteadas/no-comiteadas?

¿cómo usamos el checkpoint para hacer el recovery?

Ejemplo

Considere el siguiente redo-log después de una falla:

$$< \texttt{START}\ \mathbf{T}_1 >$$

$$< \mathbf{T}_1, A, 5 >$$

$$< \texttt{START}\ \mathbf{T}_2 >$$

$$< \texttt{COMMIT}\ \mathbf{T}_1 >$$

$$< \mathbf{T}_2, B, 10 >$$

$$< \texttt{START CKPT}(\mathbf{T}_2) >$$

$$< \mathbf{T}_2, C, 15 >$$

$$< \texttt{START}\ \mathbf{T}_3 >$$

$$< \mathbf{T}_3, E, 25 >$$

$$< \texttt{END CKPT} >$$

$$< \texttt{COMMIT}\ \mathbf{T}_2 >$$

$$< \texttt{COMMIT}\ \mathbf{T}_3 >$$

Comparación undo vs redo

- Undo logging: política **STEAL**/**FORCE**.
- Redo logging: política **NO-STEAL**/**NO-FORCE**.

¿cómo logramos una política STEAL/NO-FORCE?

Outline

Undo-redo logging

Log records:

- < `START` **T** >
- < `COMMIT` **T** >
- < `ABORT` **T** >
- < **T**, X , t_O , t_N >: donde t_O es el valor **antiguo** y t_N es valor **nuevo**.

Reglas para loggings anteriores

Regla U_1:

"Si T *modifica* X*, el log record* $<$ T, X , $t>$ *debe ser escrito antes que* X *sea escrito en disco."*

Regla U_2:

"Si T *hace commit, el log record* $<$ `COMMIT` T $>$ *debe ser escrito justo después que todos los datos modificados por* T *esten almacenados en disco."*

Regla R_1:

"Antes de modificar cualquier elemento X *en disco, es necesario que todos los logs records estén almacenados en disco, incluido el* `COMMIT` *log record."*

Reglas de undor-redo logging

Regla UR_1:

"Antes de modificar cualquier elemento en disco X, es necesario que el log record $<$ T, X , t_O , t_N $>$ este registrado en el log."

UR_1 es la "intersección" de los dos conjuntos de reglas.

Reglas de undor-redo logging

Ejemplo

T	t	Mem B_1	Mem B_2	B_1	B_2	LOG
						< START **T** >
READ(B_1, t)	1000	1000		1000	1000	
$t := t - 100$	900					
WRITE(B_1, t)		900				< **T**, B_1, 1000, 900 >
READ(B_2, t)	1000		1000			
$t := t + 100$	1100					
WRITE(B_2, t)			1100			< **T**, B_2, 1000, 1100 >
FLUSH-LOG						
UNPIN(B_1)				900		
						< COMMIT **T** >
UNPIN(B_2)					1100	

Recuperación de fallas con undo-redo logging

Después de la falla de un sistema:

- hacer **redo** de todas las transacciones que hicieron commit.
  - desde el principio hasta el final del log.

- hacer **undo** de todas las transacciones incompletas.
  - desde el final hasta el principio del log.

¿cuál hacemos primero? ¿redo-undo? ¿o undo-redo?

Recuperación de fallas con undo-redo logging

Ejemplo

T	t	Mem B_1	Mem B_2	B_1	B_2	LOG
						< START **T** >
READ(B_1, t)	1000	1000		1000	1000	
$t := t - 100$	900					
WRITE(B_1, t)		900				< **T**, B_1, 1000, 900 >
READ(B_2, t)	1000		1000			
$t := t + 100$	1100					
WRITE(B_2, t)			1100			< **T**, B_2, 1000, 1100 >
FLUSH-LOG						
UNPIN(B_1)				900		
						< COMMIT **T** >
UNPIN(B_2)					1100	

¿qué ocurre si el sistema falla en alguna instrucción?

Checkpoints en undo-redo logging

1. Escribir < `START CKPT`($\mathbf{T}_1, \ldots, \mathbf{T}_n$) > y hacer `FLUSH-LOG`.
   - $\mathbf{T}_1, \ldots, \mathbf{T}_n$ es la lista de transacciones activas y no commiteadas.
2. Escribir a disco **todas** las páginas marcadas en **dirty**.
   - Sin restringir el comienzo de nuevas transacciones.
3. Escribir < `END CKPT` > y hacer `FLUSH-LOG`.

¿cómo usamos el checkpoint para hacer el recovery?

Ejemplo

Considere el siguiente undo-redo log después de una falla:

$$< \texttt{START } \mathbf{T}_1 >$$

$$< \mathbf{T}_1, A, 4, 5 >$$

$$< \texttt{START } \mathbf{T}_2 >$$

$$< \texttt{COMMIT } \mathbf{T}_1 >$$

$$< \mathbf{T}_2, B, 9, 10 >$$

$$< \texttt{START CKPT}(\mathbf{T}_2) >$$

$$< \mathbf{T}_2, C, 14, 15 >$$

$$< \texttt{START } \mathbf{T}_3 >$$

$$< \mathbf{T}_3, D, 19, 20 >$$

$$< \texttt{END CKPT} >$$

$$< \texttt{COMMIT } \mathbf{T}_2 >$$

$$< \texttt{COMMIT } \mathbf{T}_3 >$$

Outline

Concurrencia y recuperación en caso de fallas

Undo-redo **logging**:

- No hace diferencia para schedules no-serializables.

Políticas de control de **concurrencia**:

- No se preocupan de la labor del recovery manager.

¿qué podría fallar en nuestro algoritmo de recuperación?

Ejemplo

T_1	T_2	A	B	
READ(A, x)		1000	1000	
WRITE$(A, x - 100)$		900		
	READ(A, y)			
	WRITE$(A, y * 1.1)$	990		
READ(B, x)				
WRITE$(B, x + 100)$			1100	
	READ(B, y)			
	WRITE$(B, y * 1.1)$		1210	
	COMMIT	990	1210	
ABORT				✕

Necesitamos hacer rollback en cascada!

Schedules recuperables

Definición

Un schedule S es **recuperable** si cada transacción hace `COMMIT` solo después del `COMMIT` de todas las transacción de las cuales leyó sus datos.

Ejemplo

- ¿Es este schedule recuperable?

T_1	T_2
	$w_2(A)$
$w_1(B)$	
$w_1(A)$	
	$r_2(B)$
`COMMIT`	
	`COMMIT`

Schedules recuperables

Definición

Un schedule S es **recuperable** si cada transacción hace `COMMIT` solo después del `COMMIT` de todas las transacción de las cuales leyó sus datos.

Ejemplo

- ¿y este otro schedule? ¿és recuperable?

T_1	T_2
$w_1(A)$	
$w_1(B)$	
	$w_2(A)$
	$r_2(B)$
	`COMMIT`
`COMMIT`	

Evitar rollbacks en cascadas

Definición

Un schedule S se dice que **evita rollbacks en cascadas** (ACR: *avoid cascading rollback*) si cada transacción lee solo valores escritos por transacciones que ya hicieron `COMMIT`.

Ejemplo

- ¿És este schedule del tipo ACR?

T_1	T_2
$w_1(B)$	
$w_1(A)$	
	$w_2(A)$
	$r_2(B)$
`COMMIT`	
	`COMMIT`

ACR $\Rightarrow$ recuperable.

Queremos control de concurrencia que producen schedules ACR

¿Qué políticas de control de concurrencia producen schedules ACR?

- **Strict 2PL** produce schedules ACR.
- CC basado en **timestamps** produce schedules ACR.