IIC2413

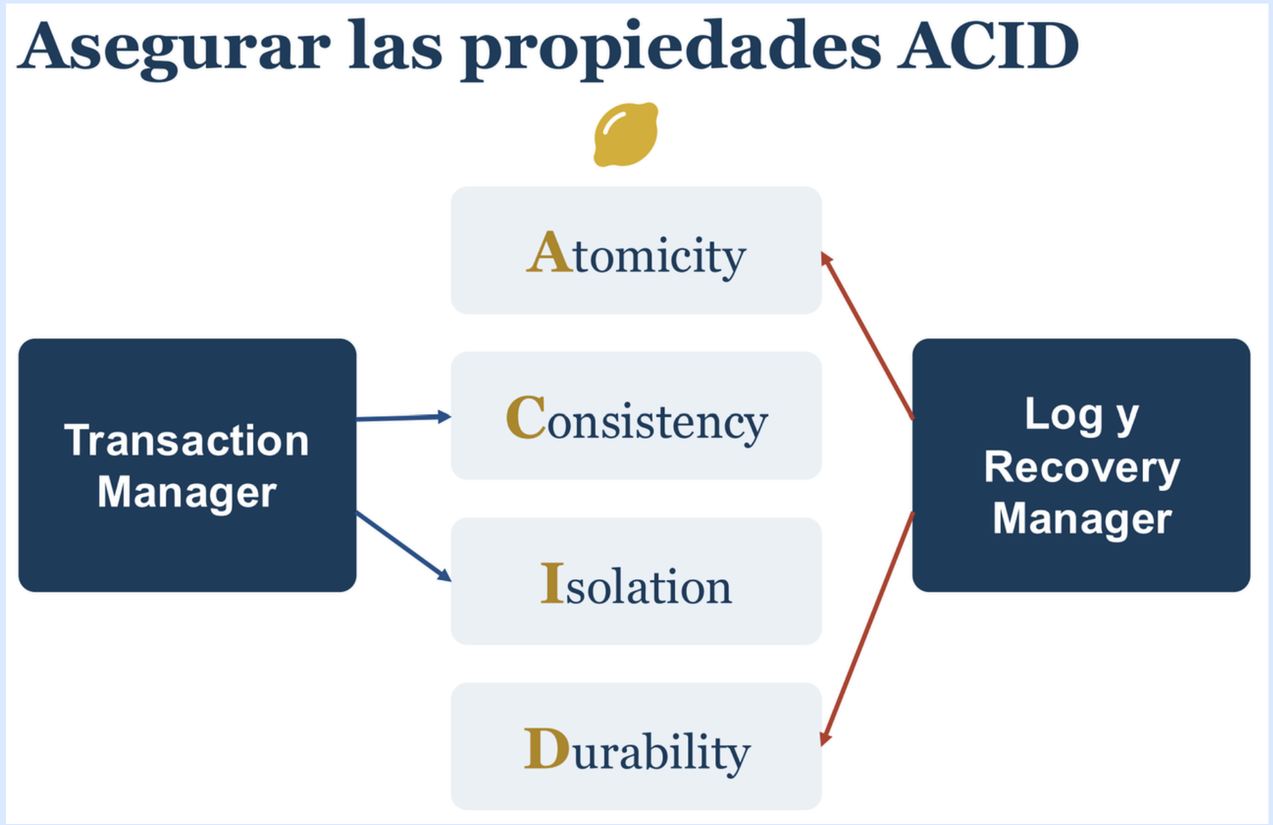
AYUDANTÍA 9

Transacciones y Fallas

TRANSACCIÓN

Secuencia de una o más operaciones que modifican o consultan la base de datos

Propósito principal: garantizar la integridad de los datos y la consistencia del estado, aún en presencia de fallos y errores.



Conflictos entre transacciones

LECTURA SUCIA

Write-Read

Una transacción accede a datos modificados por otra transacción que aún no ha terminado.

T1	T2	A
READ(A, x)		1000
WRITE(A, x – 100)		**900**
	READ(A, y) → **900**	900
	WRITE(A, y × 1.1)	990
ROLLBACK (vuelve A a 1000)		990 ❌

T1 abortó pero T2 ya trabajó con datos sucios. La BD queda inconsistente.

LECTURA NO REPETIBLE

Read-Write

Una fila se lee dos veces y su valor cambia entre ambas lecturas, porque otra transacción la modifica en ese intervalo.

T1	T2	A
READ(A, x) → 1		1
IF x > 0		1
	READ(A, y) → 1	1
	WRITE(A, y – 1)	0
	COMMIT	0
WRITE(A, x – 1) → –1		**–1** ❌

T1 razonó "A es 1, le resto 1, queda 0". Pero T2 ya lo había bajado: termina en –1, violando un check.

ACTUALIZACIÓN PERDIDA

Write-Write

Dos transacciones actualizan la misma fila en un intervalo tal que una actualización sobrescribe a la otra.

Alice	Bob	AB
READ(AB, x) → 1000		1000
	READ(AB, y) → 1000	1000
WRITE(AB, x – 100)		**900**
	WRITE(AB, y – 200)	**800** ❌

Bob escribió 1000 - 200 = 800, ignorando los -100 de Alice. La actualización de Alice se perdió: el saldo final debería ser 700.

Schedules

SCHEDULE

Secuencia de operaciones primitivas de una o más transacciones, tal que, para toda transacción, sus acciones aparecen en el mismo orden que en su definición.

Ej:

T1	T2
READ(A,x)	
x := x + 100	
WRITE(A,x)	
READ(B,x)	
x := x + 200	
WRITE(B,x)	
	READ(A,y)
	y := y * 2
	WRITE(A,y)
	READ(B,y)
	y := y * 3
	WRITE(B,y)

SCHEDULE SERIAL

Un schedule S es serial si no hay intercalación entre las acciones de sus transacciones.

T1	T2
READ(A,x)	
x := x + 100	
WRITE(A,x)	
READ(B,x)	
x := x + 200	
WRITE(B,x)	
	READ(A,y)
	y := y * 2
	WRITE(A,y)
	READ(B,y)
	y := y * 3
	WRITE(B,y)

SCHEDULE SERIALIZABLE

Un schedule S es serializable si existe algún schedule S’ serial con las mismas transacciones, tal que el resultado de S y S’ es el mismo para todo estado inicial de la BD.

T1	T2
READ(A,x)	
x := x + 100	
WRITE(A,x)	
	READ(A,y)
	y := y * 2
	WRITE(A,y)
READ(B,x)	
x := x + 200	
WRITE(B,x)	
	READ(B,y)
	y := y * 3
	WRITE(B,y)

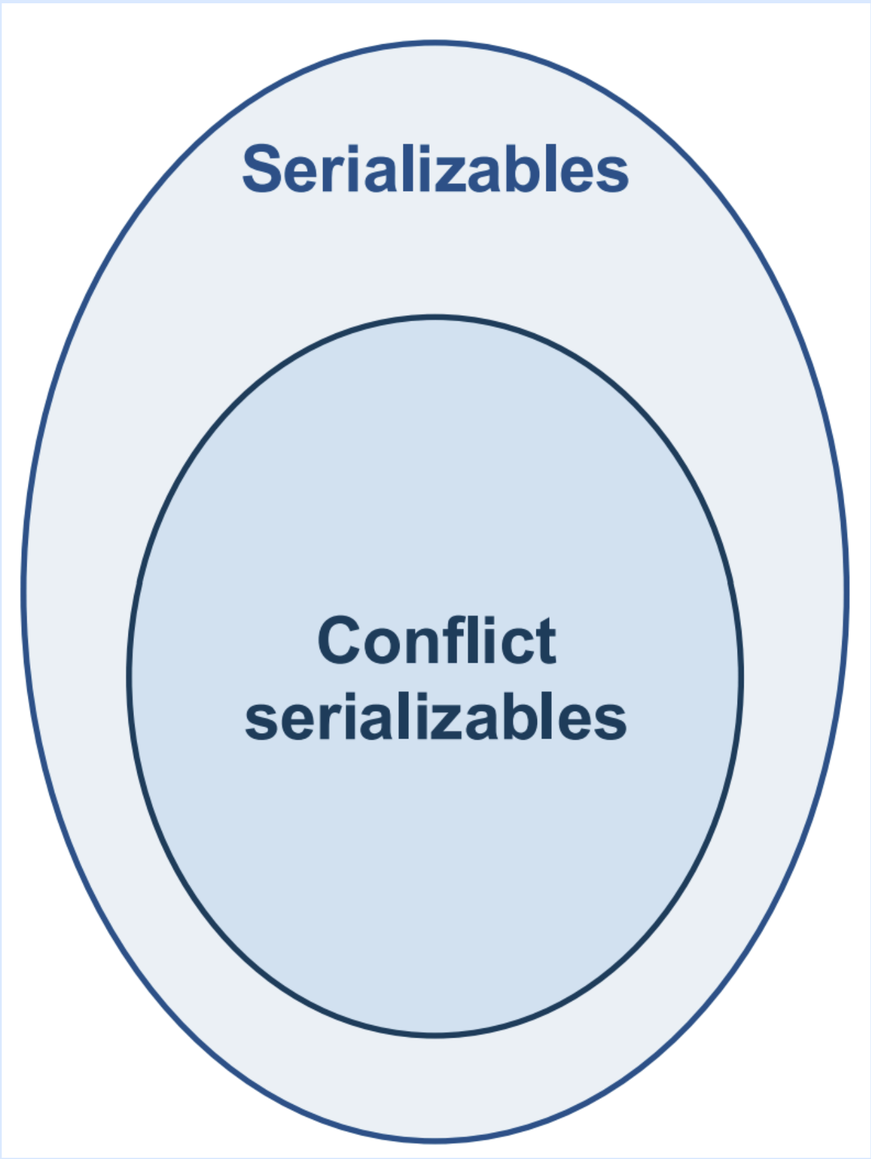
SCHEDULE NO SERIALIZABLE

Si el orden cambia, cambian los resultados; por lo mismo, el schedule no es serializable.

T1	T2
READ(A,x)	
x := x + 100	
WRITE(A,x)	
	READ(A,y)
	y := y * 2
	WRITE(A,y)
	READ(B,y)
	y := y * 3
	WRITE(B,y)
READ(B,x)	
x := x + 200	
WRITE(B,x)	

SCHEDULE CONFLICT SERIALIZABLE

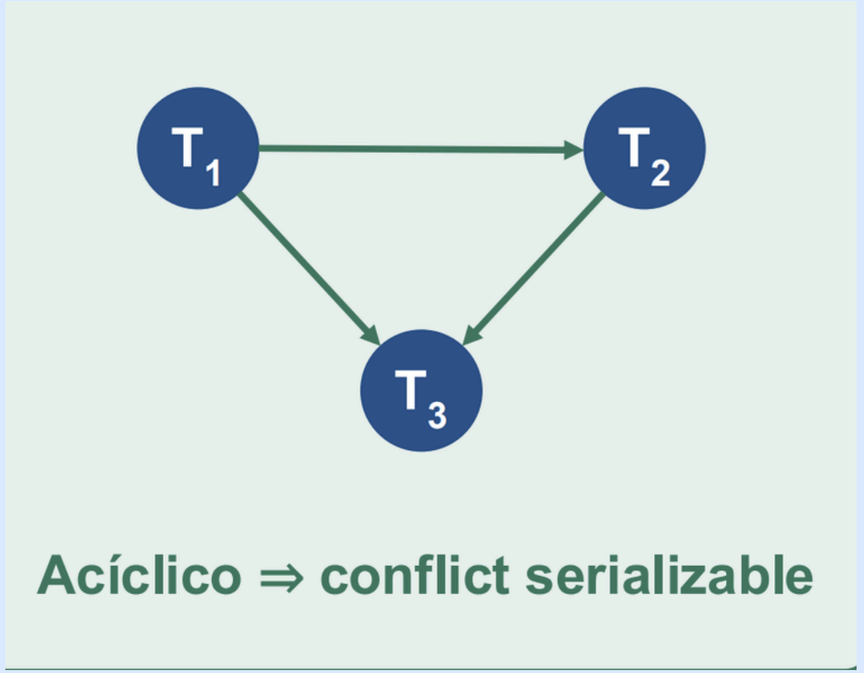
Un schedule es conflict serializable si es posible transformarlo en uno serial mediante permutaciones.

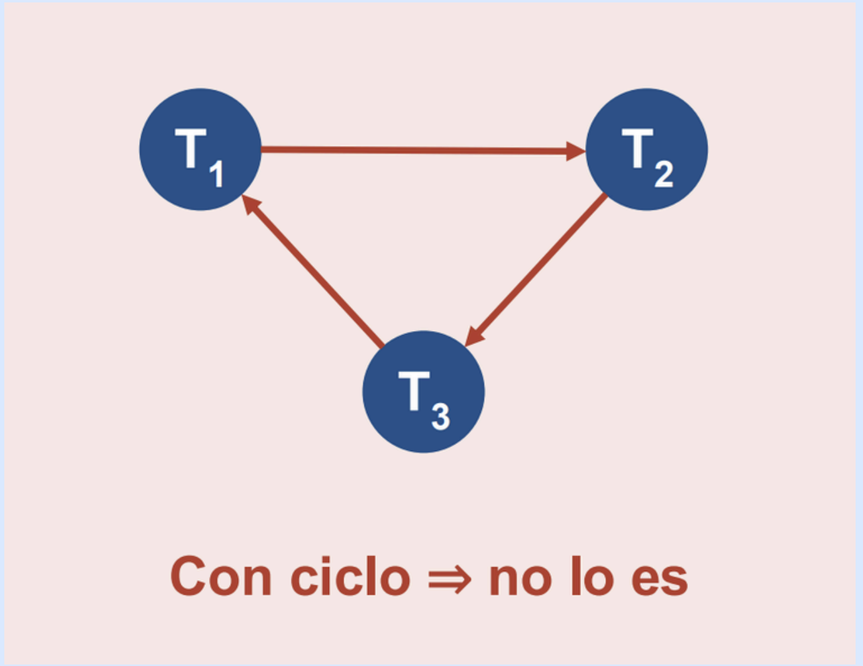


GRAFO DE PRECEDENCIA

Un schedule es conflict serializable si y solo si su grafo de precedencia es acíclico.

Un grafo de precedencia es un grafo dirigido y sin ciclos donde los nodos corresponden a instrucciones (en este caso, a transacciones).





Recuperación de Fallas

UNDO LOGGING

El registro del update es <T,X,t>, donde t es el valor antiguo de X

Regla 1: antes de escribir el nuevo valor de X en disco, primero escribir el log.

Regla 2: va después de que todos los datos que modificó T ya están en disco

Algoritmo:

1. Leer log desde el final hacia el principio
2. Si aparece <COMMIT T> o <ABORT T>, marcar T como terminada
3. Si aparece <T, X, t> y T no está terminada, restituir X = t en disco
4. Si aparece <START T>, no se deshace nada, solo indica el comienzo

UNDO LOGGING

Checkpoint completo

- Retrocedo hasta el <START CKPT (T1,...,Tn)> correspondiente
- Hago UNDO de las transacciones sin COMMIT que aparezcan desde ahí
- Todo lo anterior al <START CKPT> ya estaba OK → no necesito mirarlo

Checkpoint incompleto

- Falla ocurrió mientras se estaba haciendo el checkpoint
- Debo leer hasta inicio de transacción más antigua listada entre T1...Tn
- Hago UNDO desde ahí

REDO LOGGING

El registro del update es <T,X,v>, donde v es el valor nuevo

Regla: todo el log (incluyendo <COMMIT T>) debe estar en disco antes de escribir los datos

Algoritmo:

1. Identificar transacciones con COMMIT
2. Ignorar transacciones sin COMMIT
3. Si una transacción ya tiene END, normalmente no hay que rehacerla
4. Recorrer hacia adelante el tramo elegido del log
5. Para cada de una transacción a rehacer, escribir X = t

REDO LOGGING

Checkpoint completo

- Retrocedo hasta el <START CKPT (T1,...,Tn)> correspondiente
- Hago redo solo de las transacciones con COMMIT sin END que aparezcan desde la más antigua entre T1...Tn
- Todo lo anterior al <START CKPT> ya estaba OK → no necesito mirarlo

Checkpoint incompleto

- Falla ocurrió mientras se estaba haciendo el checkpoint
- Busco el <END CKPT> válido anterior más cercano.
- Hago REDO desde ahí

UNDO - REDO

Criterio	Undo	Redo
Registro de escritura	Valor antiguo	Valor nuevo
Lectura principal	Hacia atrás	Hacia adelante
Transacciones incompletas	**Se deshacen**	Se ignoran
Transacciones confirmadas	Se ignoran	**Se repiten**
Escribir COMMIT	Después de almacenar en disco	Antes de almacenar en disco

Ejercicios

EJERCICIO 1: I2 2026-1

b) Considere el siguiente log de Redo Logging y una falla ocurre en el punto indicado:

```
<START T1>
<START T2>
<T1, A, 100>
<T2, B, 200>
<T1, C, 150>
<COMMIT T2>
<START T3>
<T3, D, 50>
<COMMIT T1>
<T2, END>
* FALLA AQUÍ
```

i) ¿Cuáles transacciones se deben rehacer (redo) y cuáles se deben ignorar? Justifica. (3 pts)

ii) Describe el orden en que se aplica redo y los valores finales de A, B, C, D en disco. (3 pts)

EJERCICIO 1: I2 2026-1

log de Redo Logging

```
<START T1>
<START T2>
<T1, A, 100>
<T2, B, 200>
<T1, C, 150>
<COMMIT T2>
<START T3>
<T3, D, 50>
<COMMIT T1>
<T2, END>
* FALLA AQUÍ
```

1) ¿Cuáles transacciones se deben rehacer (redo) y cuáles se deben ignorar? Justifica.

EJERCICIO 1: I2 2026-1

log de Redo Logging

```
<START T1>
<START T2>
<T1, A, 100>
<T2, B, 200>
<T1, C, 150>
<COMMIT T2>
<START T3>
<T3, D, 50>
<COMMIT T1>
<T2, END>
* FALLA AQUÍ
```

i) ¿Cuáles transacciones se rehacen? (3 pts)

Regla de Redo Logging: solo se rehacen transacciones que tienen `<COMMIT T>` en el log.

- **T1: se rehace** → tiene `<COMMIT T1>`.
- **T2: se ignora** → ya tiene `<T2, END>`; esto significa que su redo ya fue aplicado y confirmado en disco, no hay nada pendiente.
- **T3: no se rehace** → no tiene `<COMMIT T3>` en el log, por lo que sus cambios nunca se garantizaron y se descartan (D no se actualiza).

EJERCICIO 1: I2 2026-1

log de Redo Logging

```
<START T1>
<START T2>
<T1, A, 100>
<T2, B, 200>
<T1, C, 150>
<COMMIT T2>
<START T3>
<T3, D, 50>
<COMMIT T1>
<T2, END>
* FALLA AQUÍ
```

II) Describe el orden en que se aplica redo y los valores finales de A, B, C, D en disco.

EJERCICIO 1: I2 2026-1

log de Redo Logging

```
<START T1>
<START T2>
<T1, A, 100>
<T2, B, 200>
<T1, C, 150>
<COMMIT T2>
<START T3>
<T3, D, 50>
<COMMIT T1>
<T2, END>
* FALLA AQUÍ
```

El recovery se procesa **de inicio a fin** del log. Se aplican en orden las escrituras de las transacciones que sí tienen COMMIT:

a) `<T1, A, 100>` → T1 tiene COMMIT más adelante → se aplicará $A = 100$.
b) `<T2, B, 200>` → T2 tiene COMMIT y END → no es necesario aplicar.
c) `<T1, C, 150>` → T1 tiene COMMIT más adelante → se aplicará $C = 150$.
d) `<T3, D, 50>` → T3 **no** tiene COMMIT → no se aplica.
e) Procesando `<COMMIT T1>` → se aplica: $A = 100$ y $C = 150$.

Valores finales en disco:

$$A = 100 \quad B = 200 \quad C = 150 \quad D = \text{sin modificar (valor anterior)}$$

EJERCICIO 2: I2 2026-1

Considere el siguiente log usando Undo Logging y una falla que ocurre en el punto indicado (5 pts):

```
<START T1>
<T1, a, 5>
<START T2>
<T2, b, 10>
<START CKPT (T1, T2)>
<T2, c, 15>
<START T3>
<T1, d, 20>
<COMMIT T1>
<T3, e, 25>
<COMMIT T2>
<END CKPT>
* FALLA AQUÍ
```

I) ¿Hasta dónde debe leer el log? (2 pts)

II) ¿Qué acciones se deshacen? Justifica usando las reglas del Nonquiescent Checkpoint. (3 pts)

EJERCICIO 2: I2 2026-1

log usando Undo Logging

```
<START T1>
<T1, a, 5>
<START T2>
<T2, b, 10>
<START CKPT (T1, T2)>
<T2, c, 15>
<START T3>
<T1, d, 20>
<COMMIT T1>
<T3, e, 25>
<COMMIT T2>
<END CKPT>
* FALLA AQUÍ
```

1) ¿Hasta dónde debe leer el log?

EJERCICIO 2: I2 2026-1

log usando Undo Logging

```
<START T1>
<T1, a, 5>
<START T2>
<T2, b, 10>
<START CKPT (T1, T2)>
<T2, c, 15>
<START T3>
<T1, d, 20>
<COMMIT T1>
<T3, e, 25>
<COMMIT T2>
<END CKPT>
* FALLA AQUÍ
```

I) ¿Hasta dónde debe leer el log?

Como se encuentra `<END CKPT>` leyendo de atrás hacia adelante, el recovery **solo necesita leer hasta el inicio de** `<START CKPT (T1, T2)>`. No es necesario ir más atrás porque todo lo anterior al checkpoint ya está garantizado en disco.

EJERCICIO 2: I2 2026-1

log usando Undo Logging

```
<START T1>
<T1, a, 5>
<START T2>
<T2, b, 10>
<START CKPT (T1, T2)>
<T2, c, 15>
<START T3>
<T1, d, 20>
<COMMIT T1>
<T3, e, 25>
<COMMIT T2>
<END CKPT>
* FALLA AQUÍ
```

II) ¿Qué acciones se deshacen?

EJERCICIO 2: I2 2026-1

log usando Undo Logging

```
<START T1>
<T1, a, 5>
<START T2>
<T2, b, 10>
<START CKPT (T1, T2)>
<T2, c, 15>
<START T3>
<T1, d, 20>
<COMMIT T1>
<T3, e, 25>
<COMMIT T2>
<END CKPT>
* FALLA AQUÍ
```

II) ¿Qué acciones se deshacen?

Leyendo de atrás hacia adelante desde la falla hasta `<START CKPT>`:

- `<COMMIT T2>` → T2 se marca como realizada.
- `<T3, e, 25>` → T3 **no** tiene COMMIT ni ABORT → **se deshace**: $e := 25$.
- `<COMMIT T1>` → T1 se marca como realizada.
- `<T1, d, 20>` → T1 ya marcada como realizada → no se deshace.
- `<T2, c, 15>` → T2 ya marcada como realizada → no se deshace.
- Se llega a `<START CKPT (T1, T2)>` → se detiene el recovery.

Conclusión: solo se deshace la escritura de e realizada por T3, ya que nunca confirmó. T1 y T2 quedan con sus valores finales porque ambas hicieron commit.