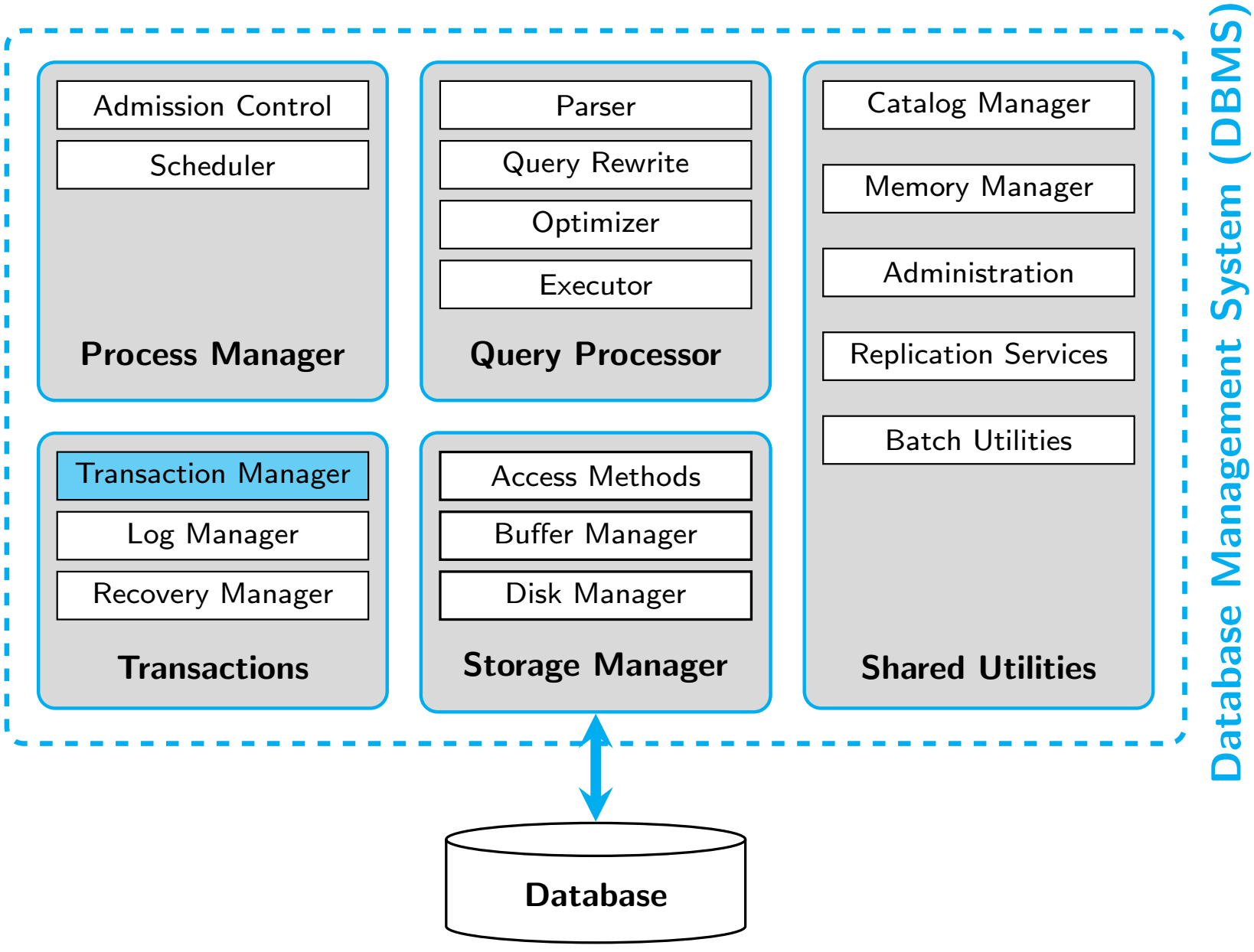
Control de concurrencia basado en locks

Clase 18

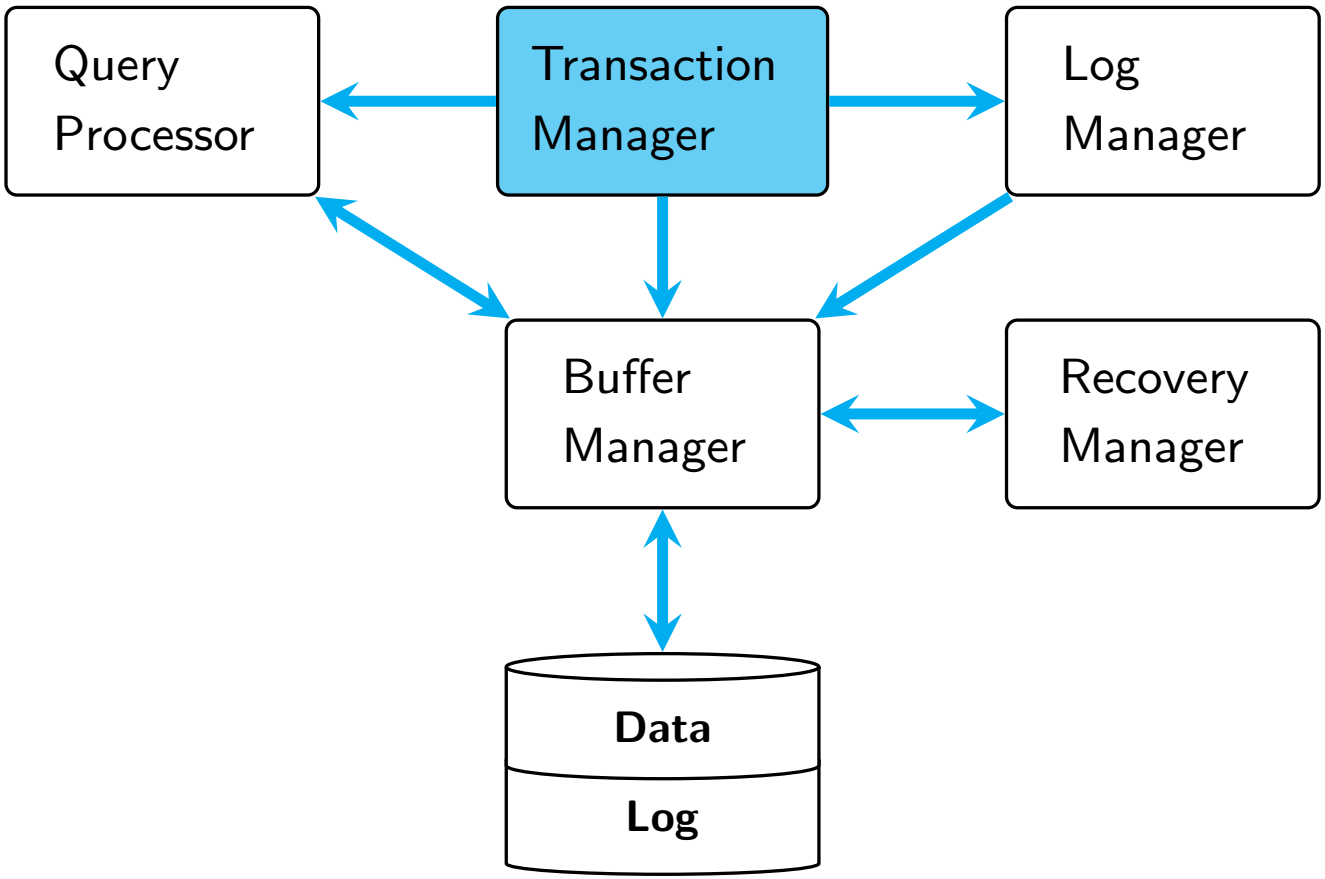
IIC 3413

Prof. Cristian Riveros

Control de concurrencia basado en locks



Zoom a la arquitectura de las transacciones



Definiciones de la clase anterior...

Definición

- Un **schedule** S es una secuencia de operaciones primitivas de una o más transacciones, tal que, para toda transacción T_i, las acciones de T_i aparecen en S en el mismo orden de su definición.

- Un schedule S es **serial** si todas las acciones de cada transacción son ejecutas en grupo y no hay intercalación de acciones.

- Un schedule S es **serializable** si existe algún serial schedule S' tal que el resultado de S y S' es el mismo para todo estado inicial de la BD.

Definiciones de la clase anterior...

Ejemplo

Un schedule **serial** para T_1 y T_2:

T_1	T_2
$r_1(A)$	
$w_1(A)$	
$r_1(B)$	
$w_1(B)$	
	$r_2(A)$
	$w_2(A)$
	$r_2(B)$
	$w_2(B)$

¿cuál es el problema con este último **schedule**?

Definiciones de la clase anterior...

Ejemplo

Un schedule **serializable** para T_1 y T_2:

T_1	T_2
$r_1(A)$	
$w_1(A)$	
	$r_2(A)$
	$w_2(A)$
$r_1(B)$	
$w_1(B)$	
	$r_2(B)$
	$w_2(B)$

¿cuál es el problema con este último **schedule**?

Definiciones de la clase anterior...

Ejemplo

Un schedule NO **serializable** para T_1 y T_2:

T_1	T_2
$r_1(A)$	
$w_1(A)$	
	$r_2(A)$
	$w_2(A)$
	$r_2(B)$
	$w_2(B)$
$r_1(B)$	
$w_1(B)$	

¿cuál es el problema con este último **schedule**?

Outline

Conflict serializable

Locks

Mas sobre locks

SQL

Outline

Conflictos entre acciones

Sea S un schedule de transacciones $T_1, \ldots, T_n$.

Las siguientes acciones consecutivas son **NO conflictivas** en S:

- $r_i(X), r_j(Y)$.
- $r_i(X), w_j(Y)$ con $X \neq Y$.
- $w_i(X), r_j(Y)$ con $X \neq Y$.
- $w_i(X), w_j(Y)$ con $X \neq Y$.

con $T_i \neq T_j$.

Conflictos entre acciones

Definición

Dos acciones t_1, t_2 son **NO conflictivas** si para todo secuencia de acciones u y v se tiene que:

$$u\ t_1\ t_2\ v \quad \text{es equivalente a} \quad u\ t_2\ t_1\ v$$

Siempre podemos **cambiar** transacciones NO conflictivas!

¿cuáles son las acciones conflictivas?

Sea S un schedule de transacciones $T_1, \ldots, T_n$.

Las siguientes acciones consecutivas decimos que son **conflictivas** en S:

- $p_i(X), q_i(Y)$ con $p, q \in \{r, w\}$.
- $r_i(X), w_j(X)$.
- $w_i(X), r_j(X)$.
- $w_i(X), w_j(X)$.

con $T_i \neq T_j$.

Estas acciones no se pueden cambiar a la ligera!

Conflictos entre acciones

Definición

- Dos acciones t_1, t_2 son **NO conflictivas** si para todo secuencia de acciones u y v se tiene que:

$$u\ t_1\ t_2\ v \quad \text{es equivalente a} \quad u\ t_2\ t_1\ v$$

- Para dos acciones t_1, t_2 **NO conflictivas** y toda secuencia de acciones u y v se define:

$$u\ t_1\ t_2\ v \vdash u\ t_2\ t_1\ v$$

- Para toda secuencia de acciones u y v decimos que $u \vdash^* v$ si $u \vdash v$ o existe una secuencia de acciones w:

$$u \vdash^* w \quad \text{y} \quad w \vdash^* v$$

Conflictos entre acciones

Ejemplo

¿cómo convertimos este schedule en un schedule serial?

T_1	T_2
$r_1(A)$	
$w_1(A)$	
	$r_2(A)$
	$w_2(A)$
$r_1(B)$	
$w_1(B)$	
	$r_2(B)$
	$w_2(B)$

Conflict serializable

Definición

Un schedule S es **conflict serializable** si existe un serial schedule S' tal que:

$$S \vdash^* S'$$

- Si S es conflict serializable entonces es serializable (¿por qué?).

¿cómo podriamos verificar que un schedule es **conflict serializable**?

¿serializable $\Rightarrow$ **conflict** serializable?

Tenemos que:

conflict serializable $\Rightarrow$ serializable.

¿es la otra dirección también verdadera?

Considere este schedule:

$$w_1(A)\ w_2(A)\ w_2(B)\ w_1(B)\ w_3(B)$$

Grafo de precedencia

Sea S un schedule de transacciones $T_1, \ldots, T_n$.

Definición

1. Para dos transacciones T_i y T_j decimos que T_i **precede** a T_j en S ($T_i <_S T_j$) si:

   - $S = u\ p_i(X)\ w\ q_j(Y)\ v$
   - $p_i(X)$ y $q_j(Y)$ son acciones **conflictivas**.

   donde u, v, y w son secuencia de acciones en S.

2. La relación $<_S$ define el grafo de precedencia.

$$\mathcal{G}_S = (\{T_1, \ldots, T_n\}, <_S)$$

Grafo de precedencia

Ejemplo

¿cuál es el grafo de precedencia de este schedule?

T_1	T_2	T_3
	$r_2(A)$	
$r_1(B)$		
	$w_2(A)$	
		$r_3(A)$
$w_1(B)$		
		$w_3(A)$
	$r_2(B)$	
	$w_2(B)$	

Grafo de precedencia

Ejemplo

¿y de este schedule?

T_1	T_2	T_3
	$r_2(A)$	
$r_1(B)$		
	$w_2(A)$	
	$r_2(B)$	
		$r_3(A)$
$w_1(B)$		
		$w_3(A)$
	$w_2(B)$	

Conflict serializable y grafo de precedencia

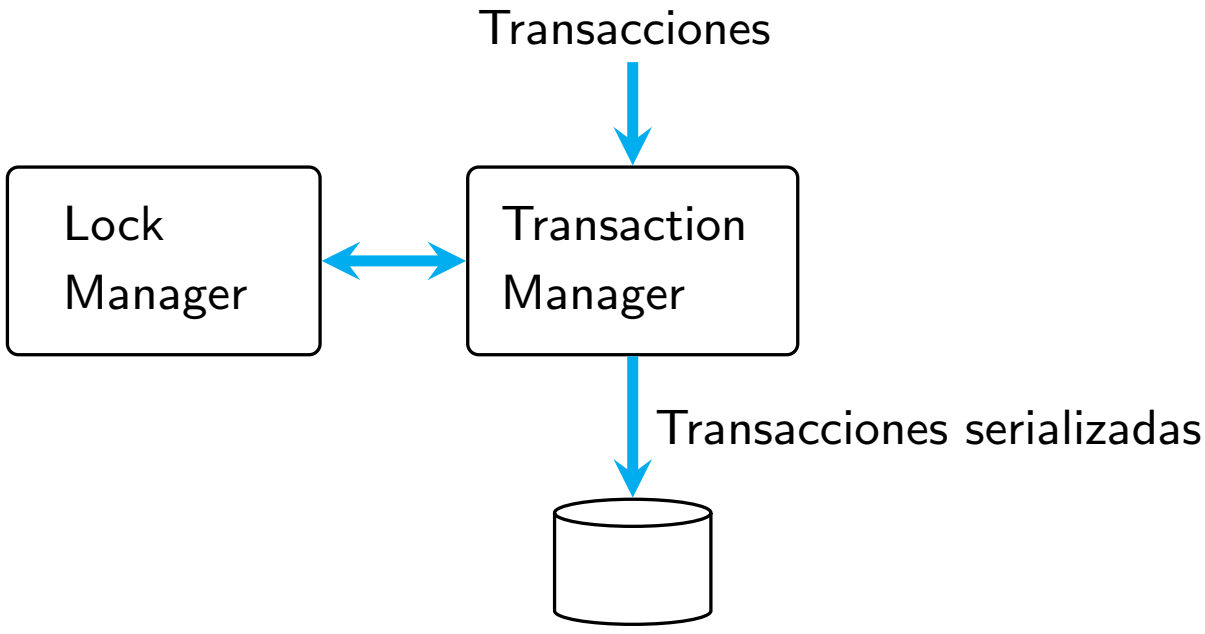
Teorema

Un schedule S es conflict serializable ssi $\mathcal{G}_S$ es **acíclico**.

¿cómo demostramos este teorema?

Outline

Locks



Lock manager (o Lock table) interfaz:

- `requestLock`(*element*).
- `releaseLock`(*element*).

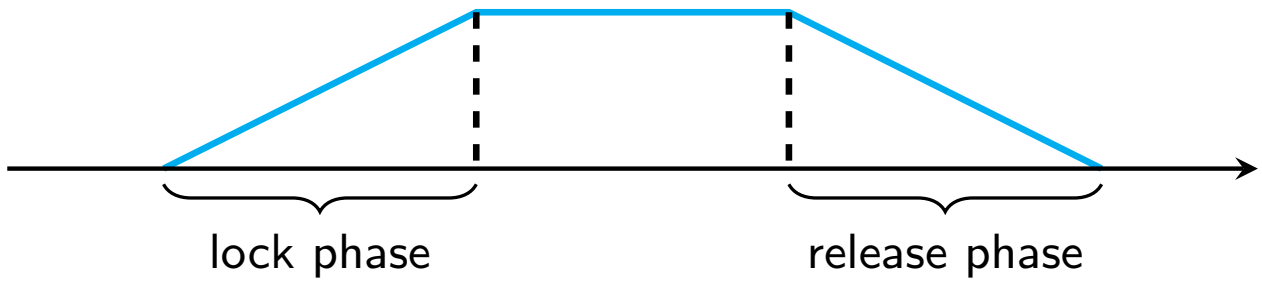
Principios de uso de locks

1. **Consistencia**:
   - Transacciones solo pueden leer o escribir un elemento si tienen su lock respectivo.
   - Si una transacción adquiere un lock, debe hacer released del lock en algún momento.
2. **Validez**: solo una transacción puede tener el lock en cualquier momento.

Two-Phase Locking (2PL)

Política de locks: toda transacción pasa por **dos fases**:

1. **Lock phase**: locks son pedidos a medida que son necesitados.
2. **Release phase**: Locks son liberados si ya no son requeridos.



"Una vez que la transacción libera algún lock,
no puede adquirir ningún lock nuevo."

Two-Phase Locking (2PL)

Ejemplo

¿cómo sería una posible ejecución de estas transacciones con 2PL?

T_1	T_2
$r_1(A)$	$r_2(A)$
$w_1(A)$	$w_2(A)$
$r_1(B)$	$r_2(B)$
$w_1(B)$	$w_2(B)$

Two-Phase Locking (2PL)

Teorema

2PL produce solo **conflict** serializable schedules.

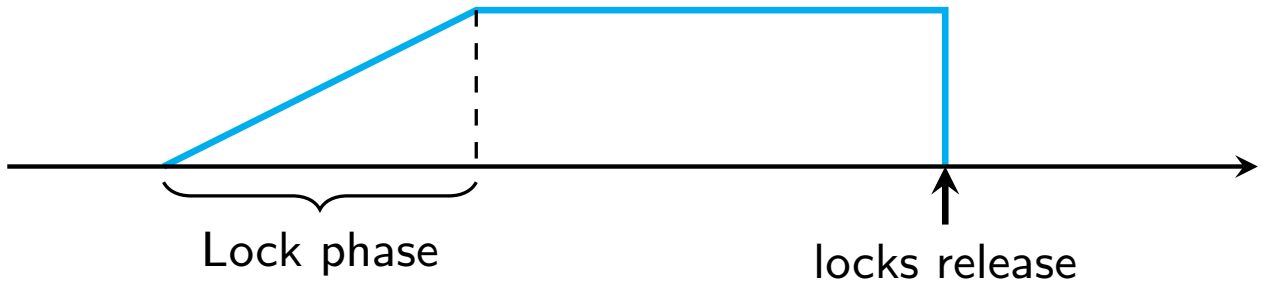
¿cómo demostramos este teorema?

¿cuáles son los problemas con 2PL?

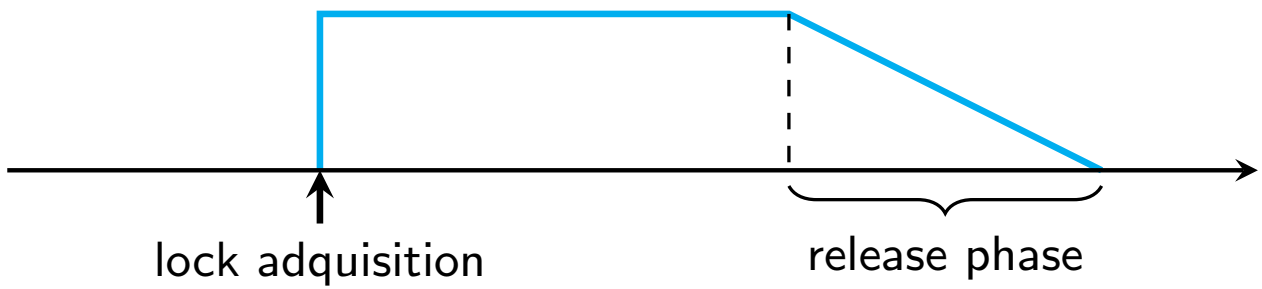
1. Cascading Rollback.
   - Solución: Strict 2PL.
2. Interferencia entre operaciones `READ`.
   - Solución: Shared y exclusive locks.
3. Deadlocks.
   - Solución: detección/prevención de deadlocks.

Variaciones de Two-Phase Locking (2PL)

Strict 2PL.

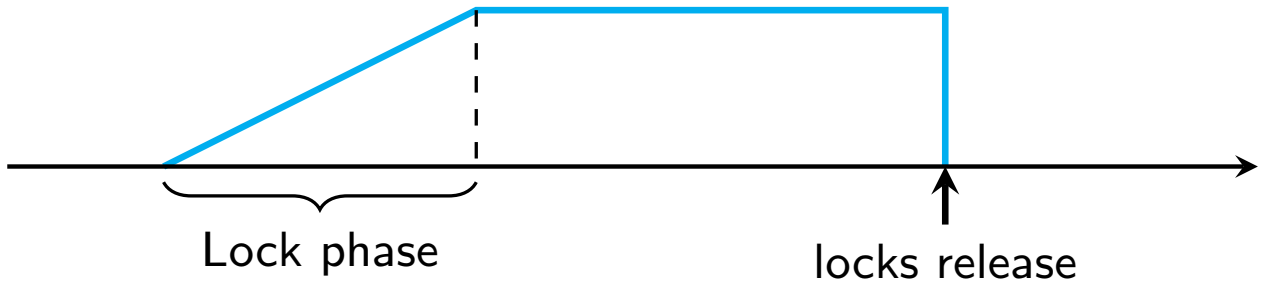


Preclaiming 2PL.



Variaciones de Two-Phase Locking (2PL)

Strict 2PL.



Ventaja de Strict 2PL:

- **NO** necesita de **Cascading Rollback**.
- Mas sencillo de implementar.

Lock modes

Dos tipos de locks:

1. **Shared** locks (slock): para leer.
2. **Exclusive** locks (xlock): para escribir.

Reglas:

- slocks pueden ser compartidos por varias transacciones.
- Una vez solicitado un xlock X de E, X debe esperar la liberación de todos los slocks o xlocks de E antes de ser otorgado.
- Una vez otorgado un xlock X de E, ningún slock o xlock de E puede ser otorgado hasta que el xlock X de E sea liberado.

¿cuáles son las ventajas de estos modos de locks?

Lock modes

Dos tipos de locks:

1. **Shared** locks (slock): para leer.
2. **Exclusive** locks (xlock): para escribir.

Reglas:

		Lock solicitado	
		S	X
Locks	S	Si	No
Actuales	X	No	No

¿cuáles son las ventajas de estos modos de locks?

Lock modes y 2PL

Teorema

2PL + lock modes produce solo **conflict** serializable schedules.

Detección de deadlocks

1. Detección de ciclos en grafo de espera.
2. Timeout.
3. Prevención de deadlocks (timestamps):
   - wait-die.
   - wound-wait.

Outline

Tuplas fantasmas / the phantom problem

Suposicion realizada hasta el momento:

Transacciones solo modifican, no hay inserts o deletes!

Tuplas fantasmas / the phantom problem

Ejemplo

- T_1 desea calcular la suma de todos los valores $D_1, \ldots, D_n$ de la relación D, almacenarlos en la tupla Y y escribir en X.
- T_2 inserta un nuevo valor en D y actualiza el valor X.

T_1	T_2
$\texttt{READ}(D_1, x)$	
$y := y + x$	
$\vdots$	
$\texttt{READ}(D_n, x)$	
$y := y + x$	
	$\texttt{INSERT}(D_{n+1}, t_1)$
	$\texttt{WRITE}(X, t_2)$
$\texttt{WRITE}(Y, y)$	
$\texttt{WRITE}(X, t_3)$	

... este schedule no es serializable!!

D_{n+1} es un valor/tupla fantasma!

Solución a tuplas fantasmas

- Uso de locks de mayor granularidad.

Granularidad de locks

Política de locks para distintos elementos:

- Atributo de una tupla.
- Tupla/record.
- Página.
- Relación.

Alternativas:

- Locking jerárquico (usado por DBMS comerciales).

Outline

Recordatorio de problemas con transacciones concurrentes

1. Conflictos Write-Read (WR): lecturas sucias.

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, y)			
	WRITE$(B, y * 1.1)$		1100	
READ(B, x)				
WRITE$(B, x + 100)$		990	1200	✗

Recordatorio de problemas con transacciones concurrentes

1. Conflictos Write-Read (WR): lecturas sucias.

2. Conflictos Read-Write (RW): lecturas irrepetibles.

T_1	T_2	A	
`READ`(A, x)		1	
`IF`$(x > 0)$			
	`READ`(A, y)		
	`IF`$(y > 0)$		
	`WRITE`$(A, y - 1)$	0	
	`ENDIF`		
`WRITE`$(A, x - 1)$		-1	
`ENDIF`		-1	✕

Recordatorio de problemas con transacciones concurrentes

1. Conflictos Write-Read (WR): lecturas sucias.
2. Conflictos Read-Write (RW): lecturas irrepetibles.
3. Conflictos Write-Write (WW): reescritura de datos temporales.

T_1	T_2	A	B	
`WRITE`$(A, 1000)$		1000		
	`WRITE`$(A, 2000)$	2000		
	`WRITE`$(B, 2000)$		2000	
`WRITE`$(B, 1000)$		2000	1000	✗

Niveles de Isolation en SQL

```
SET TRANSACTION ISOLATION LEVEL ****
```

1. SERIALIZABLE
2. REPEATABLE READ
3. READ COMMITTED
4. READ UNCOMMITTED

Niveles de Isolation en SQL

Nivel	lecturas sucias	lecturas irrep.	Fantasmas
`SERIALIZABLE`	✗	✗	✗
`REPEATABLE READ`	✗	✗	✓
`READ COMMITTED`	✗	✓	✓
`READ UNCOMMITTED`	✓	✓	✓

1. `SERIALIZABLE`: Strict 2PL y locks jerárquicos.
2. `REPEATABLE READ`: Strict 2PL y locks en tuplas.
3. `READ COMMITTED`: Uso de slocks, pero son liberados inmediatamente.
4. `READ UNCOMMITTED`: Uso de xlocks pero no de slocks.

Utilidad de distinto nivel

1. `SERIALIZABLE`
   - Mayor nivel de isolation, menor performance.
2. `REPEATABLE READ`
   - Útil para transacciones sin `INSERT`/`DELETE`.
3. `READ COMMITTED`
   - Útil para transacciones con pocas modificaciones.
4. `READ UNCOMMITTED`
   - Útil para transacciones de solo lectura o tolerante a errores.