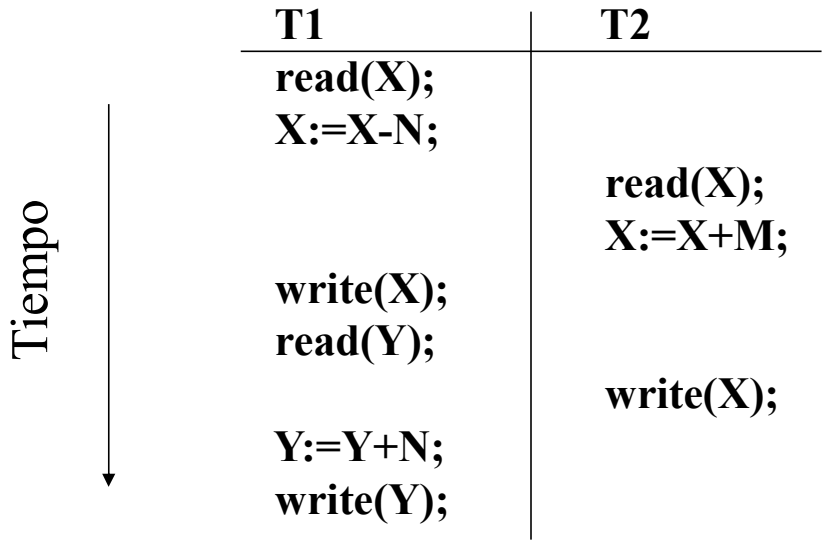
Transacciones y Concurrencia

Fundamentos de Bases de Datos

Inco – Fing - Udelar

Control de Concurrencia

- **Sobre una BD se podrían ejecutar muchos procesos concurrentemente sobre exactamente los mismos datos.**
- **Estos procesos podrían estar interfiriendo unos con otros. De qué forma?**



X queda con un valor incorrecto !

Actualización Perdida

Control de Concurrencia

- **Control de concurrencia es la coordinación de procesos concurrentes que operan sobre datos compartidos y que podrían interferir entre ellos.**

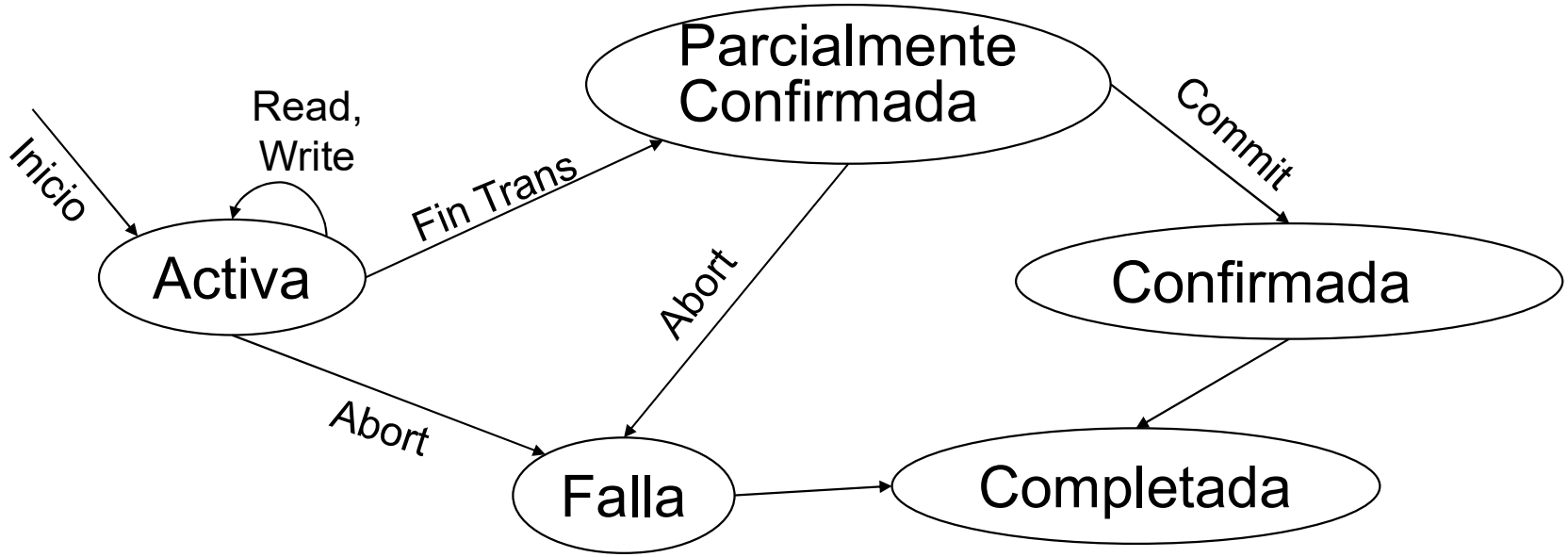
- **Temas**
  - Estudio de un modelo formal y de los principios básicos de las técnicas que debe implementar el manejador para garantizar la consistencia de la base en la situación de modificaciones concurrentes.
  - Transacción
  - Seriabilidad y Recuperabilidad
  - Protocolos de Control de Concurrencia

Transacción

- **A cada uno de los procesos concurrentes que se ejecuta sobre datos compartidos, se le llama Transacción si cumple las propiedades ACID:**
  - **Atomicidad** (Atomicity): desde el punto de vista del resultado, o se ejecuta totalmente, o no se ejecuta.
  - **Consistencia** (Consistency preservation): siempre lleva a la base de un estado consistente a otro estado consistente.
  - **Aislamiento** (Isolation): una transacción no debe interferir con otra.
  - **Durabilidad** (Durability): los cambios de una transacción confirmada deben ser permanentes en la base.

Transacciones

- **Para garantizar ACID, las transacciones tiene que pasar por determinados estados:**



Control de Concurrencia: Definiciones y Notación

- **Operaciones**
  - Para el control de concurrencia sólo interesan

$read_i(X)$, $r_i(X)$: la transacción i lee el ítem X de la base.

$write_i(X)$, $w_i(X)$: la transacción i escribe el ítem X de la base.

$commit_i$, c_i: la transacción i confirma que todas sus modificaciones deben ser permanentes en la base.

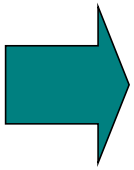
$abort_i$, a_i: la transacción i indica que ninguna de sus modificaciones debe ser permanentes en la base.

Control de Concurrencia: Definiciones y Notación

- **Rollback**
  - Es la acción de recuperar el estado anterior de la base frente a un abort de una transacción.

El Manejador de Transacciones.

T1	T2
read(X);	
	read(X);
write(X);	
read(Y);	
	write(X);
write(Y);	
	commit;
abort;	



TRANSACTION MANAGER

T_1: $r_1(x)$, $w_1(x)$, $r_1(y)$, $w_1(y)$, a_1

T_2: $r_2(x)$, $w_2(x)$, c_2

H: $r_1(x)$, $w_1(x)$, $r_2(x)$, $r_1(y)$, $w_2(x)$, $w_1(y)$, a_1, a_2

El Manejador de Transacciones (Scheduler o Transaction Manager)

- **Se encarga de la administración de las transacciones en la Base de Datos.**
- **Para eso recibe las instrucciones que los programas pretenden ejecutar y se toma la libertad de reordenarlas, sin cambiar nunca el orden relativo de los Read y Write.**
- **Puede llegar a agregar instrucciones (nunca R/W) por su cuenta.**
- **Puede llegar a demorar la ejecución de las instrucciones.**
- **Hace lo que necesite para implementar ACID hasta donde pueda.**

Control de Concurrencia: Definiciones y Notación.

- **Historia**
  - Dado un conjunto de transacciones se le llama historia o schedule a una ordenación de todas las operaciones que intervienen en las transacciones, siempre que estas aparezcan en el mismo orden que en la transacción.
- **EJ:**
  - **T_1**: $r_1(x)$, $w_1(x)$, c_1; **T_2**: $r_2(x)$, $r_2(y)$, $w_2(x)$, c_2
  - **H_1**: $r_1(x)$,$w_1(x)$, c_1, $r_2(x)$, $r_2(y)$, $w_2(x)$, c_2
  - **H_2**: $r_2(x)$, $r_1(x)$, $w_1(x)$, $r_2(y)$, c_1, $w_2(x)$, c_2

Historia Serial

Historia Entrelazada

Control de Concurrencia: Definiciones y Notación.

- **Historia Completa. Es aquella que cumple:**
  - tiene todas las operaciones de las transacciones involucradas
  - cualquier par de operaciones de la misma transacción, deben aparecer en la historia en el mismo orden que en la transacción
  - las operaciones en conflicto deben tener su orden de aparición definido en la historia

Historias Serializables y Recuperables

- **Si las transacciones se ejecutaran siempre en forma serial, entonces no habría concurrencia pero los datos siempre serían correctos.**
- **Si las historias son entrelazadas, podría suceder que queden datos erróneos que no se puedan corregir o que si una transacción aborta otra también tenga que abortar.**
- **Ej:**
  - **T_1**: $r_1(x)$, $w_1(x)$, a_1; **T_2**: $r_2(x)$, $r_2(y)$, $w_2(x)$, c_2 (o a_2)
  - **H_1**: $r_1(x)$, $w_1(x)$, $r_2(x)$, $r_2(y)$, $w_2(x)$, c_2, a_1
  - **H_2**: $r_1(x)$, $w_1(x)$, $r_2(x)$, $r_2(y)$, $w_2(x)$, a_1, a_2

Historia No Recuperable

Historia con abortos en cascada

Historias Seriales y Serializables

- **Se necesitan historias entrelazadas pero con comportamiento de seriales.**

Operaciones en Conflicto

- **Dos operaciones (r o w) están en conflicto si cumplen a la vez las siguientes condiciones:**
  - Pertenecen a distintas transacciones.
  - Acceden al mismo ítem.
  - Una es un write.

Historias Seriales y Serializables

- **Historia Serializable**
  - Es una historia que es equivalente a una historia serial con las mismas transacciones.

- **Hay varias nociones de equivalencia (o serializabilidad):**

  - **Intuitiva.** Dos historias son equivalentes si dejan la base en el mismo estado.
    - Problema: difícil de garantizar. Puede suceder de casualidad.

  - **Por conflicto.** Si tienen todas las operaciones en conflicto en el mismo orden.

  - **Por vistas.** Si cada T_i lee de las mismas $T_j,...T_n$, en H y H'.

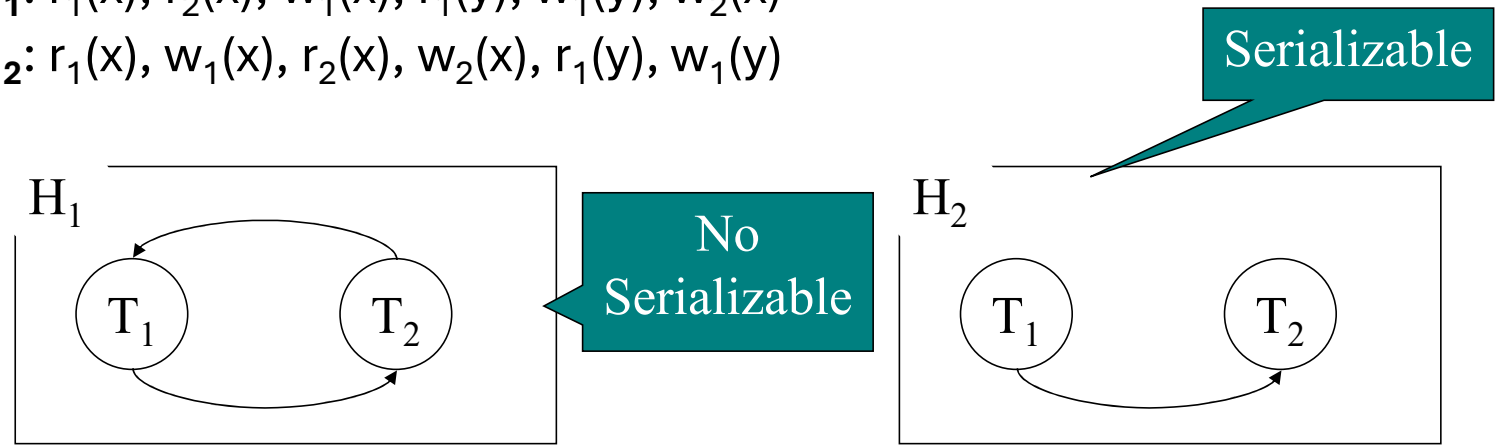
Testeo de Seriabilidad por Conflictos: Grafo de Seriabilidad

- **Construcción Grafo de Seriabilidad o Precedencia.**
  - Poner un nodo para cada transacción en la historia.
  - Si $r_j(X)$ está después de $w_i(X)$, entonces hay un arco de T_i a T_j.
  - Si $w_j(X)$ está después de $r_i(X)$, entonces hay un arco de T_i a T_j.
  - Si $w_j(X)$ está después de $w_i(X)$, entonces hay un arco de T_i a T_j.

- **Siempre se pone un arco si hay una pareja de operaciones en conflicto, desde la primera transacción a la segunda según el orden de las ops. en conflicto.**

Teorema de Seriabilidad

- **Un historia H es serializable si su grafo de seriabilidad es acíclico. [Gray-75]**
- **Ejemplo de cómo usarlo:**
  - **T_1**: $r_1(x)$, $w_1(x)$, $r_1(y)$, $w_1(y)$; **T_2**: $r_2(x)$, $w_2(x)$
  - **H_1**: $r_1(x)$, $r_2(x)$, $w_1(x)$, $r_1(y)$, $w_1(y)$, $w_2(x)$
  - **H_2**: $r_1(x)$, $w_1(x)$, $r_2(x)$, $w_2(x)$, $r_1(y)$, $w_1(y)$



Lectura de una transacción

- **T_1 lee de T_2 en H si $w_2(X)$ está antes de $r_1(X)$ y entre medio:**
  - no hay otro $w_j(X)$ (que no es abortado antes de $r_1(X)$)
  - no está a_2

Historias Recuperables

- **Una historia es Recuperable si ninguna transacción confirma hasta que confirmaron todas las transacciones desde las cuales leyó ítems. (Los *commits* están en el mismo orden que el flujo de datos)**

Historias que Evitan Abortos en Cascada

- **Una historia Evita Abortos en Cascada (EAC) si ninguna transacción lee de transacciones no confirmadas. (Los *commits* tienen que estar antes de los *reads* siguientes)**

Historias Estrictas

- **Una historia es Estricta si ninguna transacción lee o escribe hasta que todas las transacciones que escribieron ese ítem fueron confirmadas. (Los *commits* tienen que estar antes de los *reads* o *writes* siguientes).**

Historias Recuperables, EAC y Estrictas

- **Ejemplos**

  - **T_1: $w_1(x)$, $w_1(z)$, $r_1(y)$, $w_1(y)$,c_1**
  - **T_2: $r_2(y)$, $w_2(y)$, $w_2(z)$, c_2**

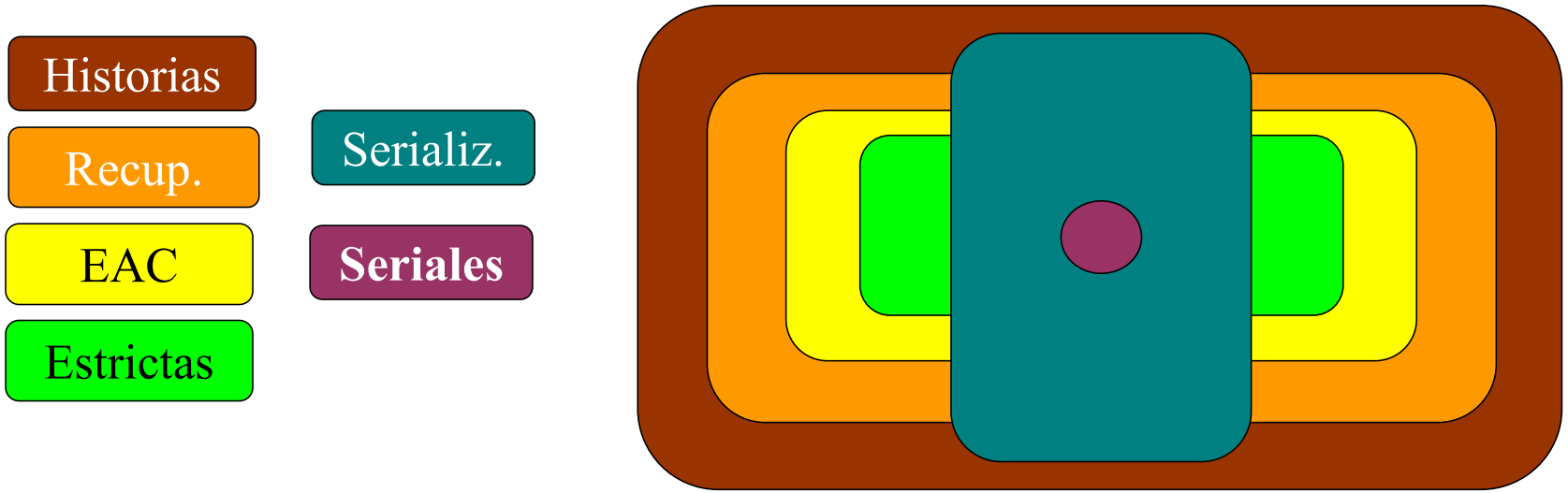
  - **H_1**: $r_2(y)$, $w_2(y)$, $w_1(x)$, $w_2(z)$, $w_1(z)$, $r_1(y)$, $w_1(y)$, c_1, c_2

  - **H_2**: $r_2(y)$, $w_2(y)$, $w_1(x)$, $w_2(z)$, $w_1(z)$, $r_1(y)$, $w_1(y)$, c_2, c_1

  - **H_3**: $r_2(y)$, $w_1(x)$, $w_2(y)$, $w_2(z)$, $w_1(z)$, c_2, $r_1(y)$, $w_1(y)$, c_1

  - **H_4**: $r_2(y)$, $w_1(x)$, $w_2(y)$, $w_2(z)$, c_2, $w_1(z)$, $r_1(y)$, $w_1(y)$, c_1

Historias Recuperables, EAC y Estrictas



Control de seriabilidad y recuperabilidad

- **El manejador debería garantizar la construcción de historias serializables, y que sean recuperables y que no tengan abortos en cascada.**
- **La forma básica de hacer esto, es demorar las operaciones en conflicto con otras anteriores hasta que estas terminen.**

- **Hay dos formas básicas de hacer esto:**
  - *Locking*
  - *Timestamps*

Lock Binario

- **Se consideran dos operaciones nuevas en una transacción T_i: $lock_i(X)$ (o $l_i(X)$) y $unlock_i(X)$ (o $u_i(X)$).**
- **Cuando se ejecuta $l_i(X)$, el DBMS hace que cualquier $l_j(X)$ (de otra transacción T_j) no termine hasta que T_i ejecute $u_i(X)$.**
- **El comportamiento es similar a los semáforos utilizados en los sistemas operativos.**
- **Ej:**
  - **T_1**: $l_1(x), r_1(x), w_1(x), u_1(x), l_1(y), r_1(y), w_1(y), u_1(y)$
  - **T_2**: $l_2(x), r_2(x), w_2(x), u_2(x)$
  - **H_1**: $l_1(x), r_1(x), w_1(x), u_1(x), l_2(x), r_2(x), w_2(x), u_2(x)\ l_1(y), r_1(y), w_1(y), u_1(y)$
  - ~~**H_2**: $l_1(x), r_1(x), l_2(x), w_1(x), u_1(x), r_2(x), w_2(x), u_2(x)\ l_1(y), r_1(y), w_1(y), u_1(y)$~~

Lock Binario

- **Lock Binario**
  - Son los anteriores. Un ítem o está bloqueado o desbloqueado.
  - Ventaja: Fácil de implementar.
  - Desventaja: Niega incluso la lectura a otras transacciones, cuando esto no sería necesario.

Read/Write Lock

- **Read/Write Lock**
  - Hay tres operaciones a considerar: $read\_lock_i(X)$ (o $rl_i(X)$), $write\_lock_i(X)$ (o $wl_i(X)$), $unlock_i(X)$ (o $u_i(X)$).
  - Ventaja: Se permite que varias transacciones hagan un rl (pero no un wl) simultáneamente sobre el mismo ítem.
  - Desventaja: Un poco más complicado de implementar.

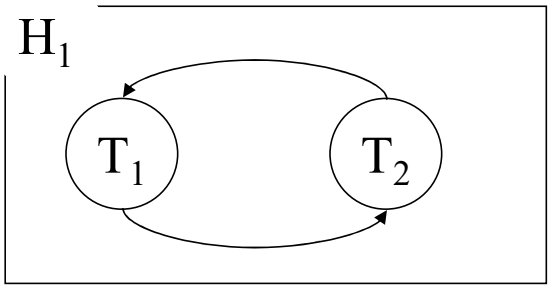
Reglas de Uso de los Locks

- **Para locks tipo binario.**
  - Antes de que T_i ejecute $r_i(X)$ (o $w_i(X)$) debe ejecutar $l_i(X)$ y luego de $r_i(X)$ (o $w_i(X)$) debe ejecutar $u_i(X)$.
  - Nunca va a ejecutar $l_i(X)$ si ya lo tiene.
- **Para locks tipo read/write**
  - Antes que T_i ejecute $r_i(X)$, debe ejecutar $rl_i(X)$ o $wl_i(X)$ y luego de $r_i(X)$ debe ejecutar $u_i(X)$.
  - Antes que T_i ejecute $w_i(X)$, debe ejecutar $wl_i(X)$ y luego de $w_i(X)$ debe ejecutar $u_i(X)$.
  - Nunca va a ejecutar algún lock sobre un ítem si ya tiene ese mismo lock sobre ese ítem.
  - Algunas veces los locks se pueden promover o degradar.

Protocolos de Locking

- **Las reglas de uso no alcanzan para garantizar seriabilidad**
  - Ej:
    - **T_1**: $rl_1(Y), r_1(Y), u_1(Y), wl_1(X), r_1(X), w_1(X), u_1(X), c_1$
    - **T_2**: $rl_2(X), r_2(X), u_2(X), wl_2(Y), r_2(Y), w_2(Y), u_2(Y), c_2$

    - **H_1**: $rl1(Y), r_1(Y), u_1(Y), rl_2(X), r_2(X), u_2(X), wl_2(Y), r_2(Y), w_2(Y), u_2(Y), wl_1(X), r_1(X), w_1(X), u_1(X), c_1, c_2$



Protocolos de Locking

- **La idea de un protocolo de locking es definir un conjunto de reglas de uso de Locks que sean más fuertes que las anteriores y sí garanticen la seriabilidad.**

Protocolos de Locking: 2PL

- **En Two Phase Locking (2PL) hay dos fases en una transacción:**
  - Fase de crecimiento o expansión: se crean locks
  - Fase de contracción: se liberan los locks

- **Ej:**

	2PL
• **T_1**: $rl_1(Y), r_1(Y), u_1(Y), wl_1(X), r_1(X), w_1(X), u_1(X), c_1$	X
• **T'_1**: $rl_1(Y), r_1(Y), wl_1(X), u_1(Y), r_1(X), w_1(X), u_1(X), c_1$	✓
• **T_2**: $rl_2(X), r_2(X), u_2(X), wl_2(Y), r_2(Y), w_2(Y), u_2(Y), c_2$	X
• **T'_2**: $rl_2(X), r_2(X), wl_2(Y), u_2(X), r_2(Y), w_2(Y), u_2(Y), c_2$	✓

Cualquier historia entre T'_1 y T'_2 va a ser **serializable**

Protocolos de Locking: 2PL Básico y Conservador

- **Hay varias versiones de 2PL con ventajas y desventajas propias. Todos garantizan serializabilidad de las historias.**
  - **2PL Básico**: Sólo exige las 2 fases
    - Ventaja: Muy simple de implementar
    - Desventaja: Susceptible a Deadlock
  - **2PL Conservador**: Exige que todos los locks se hagan antes del comienzo de la transacción.
    - Ventaja: No susceptible a Deadlock
    - Desventaja: Exige la predeclaración de todos los ítems que se van a leer o grabar (read-set, write-set), lo que no siempre es posible.

Protocolos de Locking: 2PL Estricto y Riguroso

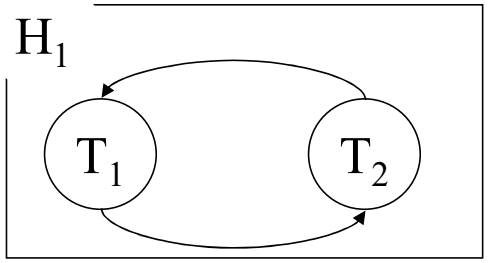
- **2PL Estricto**: Exige que no libere ningún write lock hasta después de terminar la transacción (luego del commit).
  - Ventaja: Garantiza historias estrictas.
  - Desventaja: Susceptible a Deadlock

- **2PL Riguroso**: Exige que no libere ningún (read o write) lock hasta después de terminar la transacción (luego del commit).
  - Ventaja: Garantiza historias estrictas. Más simple de Implementar que el estricto.
  - Desventaja: Susceptible a Deadlock

- **Hay protocolos viables que no sean suceptibles a Deadlock? Hay otros problemas?**

Problemas con Locks: Deadlock

- **Dos o más transacciones esperan unas por otras.**
  - Ej:
    - **T_1**: $rl_1(Y), r_1(Y), wl_1(X), u_1(Y), u_1(X), \ldots$
    - **T_2**: $rl_2(X), r_2(X), wl_2(Y), u_2(X), u_2(Y), \ldots$
    - **H_1**: $rl_1(Y), r_1(Y), rl_2(X), r_2(X), wl_1(X), wl_2(Y)$



  - Chequeo: construir grafo de espera
    - Tiene un arco de T_i a T_j si T_i está tratando de lockear un ítem que T_j tiene lockeado.
    - Si tiene ciclos, entonces hay deadlock.

Soluciones para Deadlocks

- **Protocolos con prevención de Deadlocks**
  - 2PL Conservador: NO Funciona en forma práctica.
  - Basados en TimeStamp:
    - TS(T) es un identificador para cada T que cumple que $TS(T_i) < TS(T_j)$ si T_i empezó antes que T_j.
    - Asumiendo que T_i quiere lockear un ítem que T_j ya lo tiene.
    - **Wait-die:** si $TS(T_i) < TS(T_j)$, entonces T_i está autorizada a esperar. Si $TS(T_j) < TS(T_i)$, entonces T_i aborta y es recomenzada más tarde con el mismo timestamp.
    - **Wound-wait**: si $TS(T_i) < TS(T_j)$, entonces T_j aborta y es recomenzada más tarde con el mismo timestamp. De lo contrario, T_i espera.
    - Pueden producir abortos y reinicios innecesarios.

Soluciones para Deadlocks

- **Detección**
  - Mantener el grafo de espera y si hay deadlock, “seleccionar una víctima” y hacer que aborte.
  - Útil para transacciones chicas (que trabajan sobre pocos ítems) por poco tiempo.
  - Si las transacciones son largas y trabajan sobre muchos ítems, se genera mucho overhead. Sería mejor usar prevención.

Soluciones para Deadlocks

- **TimeOut**
  - Si una transacción espera por mucho tiempo, el sistema la aborta sin importar si hay deadlock o no.

Problemas en Locks: Starvation

- **Una transacción no puede ejecutar ninguna operación por un período indefinido de tiempo.**
- **Esto puede suceder por ejemplo, por un mecanismo de "selección de víctima" equivocado.**
- **Soluciones similares a las usadas en SO:**
  - FIFO
  - Manejo de prioridades.
- **Wound-wait y Wait-die evitan este problema.**

Control de Concurrencia Basado en Ordenación de TimeStamps

- **Cada transacción T tiene un timestamp TS(T)**

- **Cada ítem X, tiene 2 timestamps:**
  - Read_TS(X): el timestamp de la transacción más joven que leyó el ítem. (El timestamp más grande).
  - Write_TS(X): el timestamp de la transacción más joven que grabó el ítem.

- **La idea consiste en ejecutar las lecturas y escrituras de acuerdo al orden de los timestamps. Si el orden se viola, entonces se aborta la transacción.**

Ordenamiento Básico por Timestamp

- **Si T trata de ejecutar un w(X)**
  - Si Read_TS(X) > TS(T) o Write_TS(X) > TS(T), T aborta y luego se reinicia con un nuevo TS. (Una transacción más joven hizo una operación en conflicto)
  - Si no, se ejecuta el w(X) y se cambia Write_TS(X) por TS(T).

- **Si T trata de ejecutar un r(X)**
  - Si write_TS(X) > TS(T), T aborta y luego se reinicia con un nuevo TS.
  - si no, se ejecuta el r(X) y se calcula el nuevo Read_TS(X) como el máximo de TS(T) y el Read_TS(X) actual.

Control de Concurrencia Multiversión

- **Algoritmos anteriores, demoran lecturas en forma innecesaria por locking o por abortos sucesivos de una transacción.**

- **Una Solución: Mantener varias versiones de cada ítem y elegir la correcta.**
  - Ventaja: Las lecturas no tienen porqué esperar.
  - Desventaja: Se necesita espacio extra para almacenar las versiones de cada ítem. Podría no ser demasiado relevante de acuerdo a los esquemas de recuperación que se utilicen.

Control de Concurrencia Multiversión

- **Idea básica: Antes de cualquier modificación a un ítem X, se guarda una versión X_i del mismo, generando para cada ítem una sucesión de versiones (X_1, X_2, ...X_k).**
- **Cada versión está formada por un valor de X, el Read_TS(X) y el Write_TS(X) asociados a ese valor.**
- **Hay dos reglas para la seriabilidad:**
  - Sea X_i la versión de X con máximo Write_TS.
    - w(X): Si T tal que Write_TS(X_i) ≤ TS(T) < Read_TS(X_i), T aborta. En cualquier otro caso se crea X_j con Read_TS(X_j)= Write_TS(X_j)= TS(T).
    - r(X): Se busca X_i tal que Write_TS(X_i) es el mas grande menor o igual que TS(T), se devuelve el valor de X_i a T y se asigna a Read_TS(X_i) el máximo de TS(T) y Read_TS(X_i).

Consideraciones sobre la Granularidad.

- **Que es un ítem ?**

- **Un Ítem puede ser:**
  - Una tupla (registro) de la una tabla de la base.
  - El valor de un atributo de una tupla.
  - Una tabla entera.
  - Un bloque de disco.
  - La base completa.

Consideraciones sobre la Granularidad

- **No es lo mismo trabajar a cualquier nivel de granularidad**
- **Cuanto mayor es el grado de granularidad, menor es el nivel de concurrencia permitido.**
  - Porque se restringe el acceso a “las partes” de los objetos.
- **Cuanto menor es el grado de granularidad, mayor es el overhead que se impone en el sistema.**
  - Porque hay que llevar el control mayor cantidad de timestamps o locks.
- **Si las transacciones tipo acceden a pocos registros, mejor manejar granularidad baja.**
- **Si acceden a muchos de una misma tabla, mejor manejar granularidad alta.**

Concurrencia en la Inserción y Eliminación.

- **Eliminación:**
  - Locking: se lockea en forma exclusiva
  - Timestamp: hay que garantizar que ninguna transacción posterior a la que elimina el registro haya leído o escrito el elemento.

- **Inserción:**
  - Locking: Se crea un lock, se inserta un registro y se libera en el momento adecuado según el protocolo.
  - TimeStamp: Se asigna el TS(X) como Write_TS y como Read_TS(X).

Inserción y Registros Fantasmas.

- **Una transacción T inserta un registro X**
- **T' accede a registros según una condicion que X cumple.**
- **Si T' lockea los registros antes que T inserte X, entonces X se convierte en un registro fantasma porque T' no se entera de su existencia a pesar de que es de ínterés para T'.**
- **Solución: Lockeo sobre índices. [Elmasri-Navathe]**