Dpto. Ciencias e Ingeniería de la Computación
Universidad Nacional del Sur

Elementos de Bases de Datos

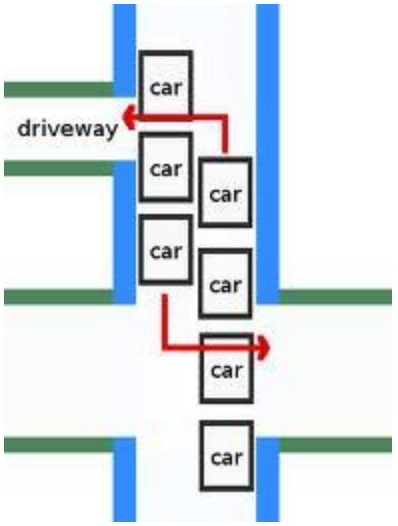
Segundo Cuatrimestre 2015

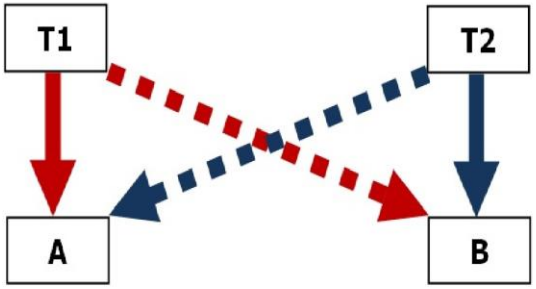
Clase 19:

Deadlock – Granularidad Múltiple – Niveles de Aislamiento

Mg. María Mercedes Vitturini
[mvitturi@uns.edu.ar]

Gestión de Deadlock





Estrategias para “manejar” deadlocks

Protocolo	Basado	¿Deadlock?	Característica
Bloqueo de Dos Fases (B2F)	Bloqueos	Si	Fase crecimiento y decrecimiento
B2F Estricto	Bloqueos	Si	Mantiene locks-x hasta el final de la transacción
B2F Riguroso	Bloqueos	Si	Mantiene locks-s y locks-x hasta el final de la transacción
B2F Refinado	Bloqueos	Si	Incluye upgrade y downgrade
Árbol	Bloqueos	No	Solo locks-x
Protocolo de Estampilla (PHE)	Estampilla	No	Ordena por hora de entrada.
PHE + Regla de Thomas	Estampilla	No	Evita retrocesos por escrituras obsoletas
Validación	Estampilla	No	Protocolo optimista. Demora los controles
Multiversión	Estampilla	No	Las transacciones de lectura nunca retroceden

Deadlock

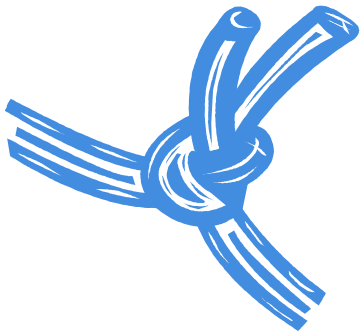
DEADLOCK O INTERBLOQUEO: es un estado indeseable que se puede alcanzar bajo ciertos de los protocolos de control de concurrencia que incluyen bloqueos + esperas.

- "Una transacción que necesita acceder a un dato Q solicita el *lock* al gestor de control de concurrencia. Este se lo concederá únicamente si su *solicitud es compatible* con los locks asignados a otras transacciones sobre el mismo dato Q. Si la solicitud no es compatible, la transacción deberá esperar."

- Puede darse que varias transacciones se esperen unas a otras sin que ninguna pueda seguir: *deadlock.*

¿Qué responsabilidad tiene el DBMS?

- prevenir estas situaciones, ó
- dejar que ocurran y accionar para superarlas.



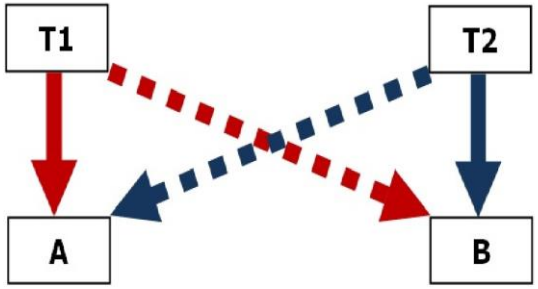
Interbloqueo o Deadlock

Consideremos las siguientes transacciones:

T_1: write (A); write (B);

T_2: write (B); write (A);

y la planificación concurrente:



T_1	T_2
Lock-X (A)	
Write (A)	
	Lock-X (B)
	Write(B)
	Espera por Lock-X (A)
Espera por Lock-X (B)	

Interbloqueo o Deadlock

Definición: un sistema está en estado de *deadlock* si existen dos o más transacciones, tal que toda transacción del conjunto está esperando por un dato que tiene retenido otra transacción del conjunto.

- Ninguna transacción del grupo puede progresar y es posible que se sumen nuevas transacciones.
- Los protocolos de bloqueos de dos fases vistos y que no imponen orden sobre los datos tienen posibilidades de generar situaciones de deadlock.

Manejo de Deadlock

- Dos estrategias:
  - **Prevención** (política *pesimista*): asegura que el sistema nunca genere un estado de deadlock.
  - **Detección** (política *optimista*): permite que el sistema eventualmente alcance una situación de deadlock. Periódicamente se corren procesos para verificar si se produjo esta situación y corregirla.

Prevención de Deadlock

- **Estrategia #1:** Exigir a la transacción que obtenga los locks de todos los ítems de datos que requiera antes iniciar su ejecución o hacerla retroceder y eliminar las esperas reteniendo recursos.
  - Antes de que comience cada transacción, es difícil predecir que Ítems va a requerir.
  - Limita la utilización de recursos.
  - Tiene problemas de inanición.
- **Estrategia #2:** Imponer un orden parcial sobre los ítems de datos y exigir a las transacciones que respeten el orden para obtener los mismos. Esta estrategia es usada por el protocolo de árbol visto.

Prevención de Deadlock con Estampillas

Estrategia #3: combinar locks y estampillas de tiempo.

- A cada transacción se le asigna **una estampilla de tiempo única** la primera vez que ingresa al sistema.
- La transacción que solicita un lock sobre un ítem de dato Q y que potencialmente pueden generar deadlock *espera o retrocede* según una de estas dos políticas:
  - **Wait-Die:** Si T_i requiere un dato que tiene T_j, T_i espera si $ts(T_i) < ts(T_j)$ (T_i es más antigua que T_j). De lo contrario, T_i es retrocedida.
  - **Wound-Wait:** Si T_i requiere un dato que tiene T_j, T_i espera si $ts(T_i) > ts(T_j)$ (T_i es más joven que T_j). De lo contrario, T_j es retrocedida y sus recursos apropiados por T_i.

Estrategias de Prevención con Estampillas

Wait-Die – Espera la más antigua

- Una transacción puede tener que esperar varias unidades de tiempo para conseguir un recurso.

Wound-Wait – Retrocede la más joven

- Puede tener menos retrocesos que wait-die.
- Las transacciones más viejas tienen precedencia sobre las nuevas y de esta manera se evitan problemas de inanición.

- Tanto en el esquema *wait-die* como en *wound-wait*, una transacción que retrocede reingresa con su estampilla de tiempo original.

Estrategias de Prevención con Timeout

- **Estrategia #4:** basado en *timeout.* Una transacción espera por obtener un lock hasta cierta cantidad de tiempo predeterminado. Transcurrido el tiempo máximo, la transacción se debe retroceder.
  - 👍 Esta política previene deadlocks.
  - 👍 Es simple de implementar;
  - 👎 Puede causar problemas de inanición.
  - 👎 Es difícil determinar un valor de timeout adecuado.

Deadlock: detección + recuperación

- No usa ninguna estrategia que evite deadlocks.
- Periódicamente, el sistema controla si existe una situación de deadlock. Si la encuentra inicia un proceso de recuperación.
- Para esto el sistema necesita:
  1. Mantener información acerca de la asignación de locks sobre ítems de datos a las transacciones así como de las solicitudes demoradas.
  2. Proveer un algoritmo de detección para determinar si el sistema está en deadlock.
  3. Romper el estado de deadlock cuando el sistema detecta que existe tal situación.

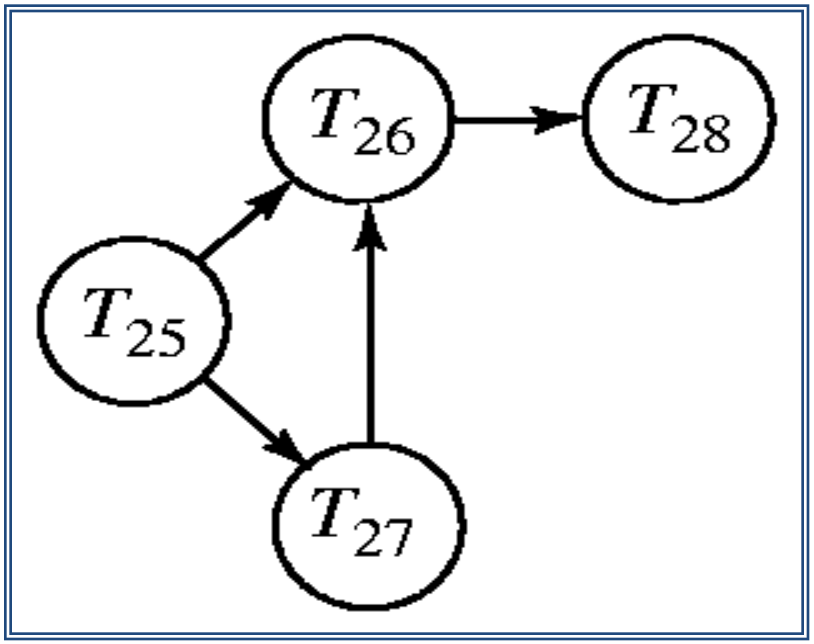
Detección + Recuperación

Mantener información

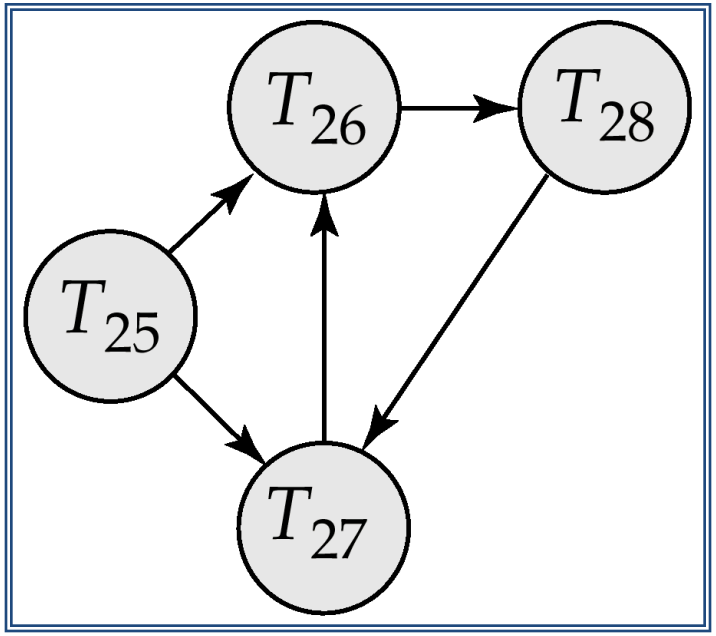
- Mantener un grafo de espera **G=(V, A)**
  - V (vértices) representan las transacciones del sistema.
  - A (arcos) es un par ordenado (T_i ;T_j). Si existe $T_i \rightarrow T_j$ en G, significa que T_j espera que T_i libere un recurso.
- Si T_j solicita un recurso que tiene T_i y que por diferencia de compatibilidades debe esperar que lo libere, se agrega $T_i \rightarrow T_j$.
- Cuando T_i libera el recurso, el arco se remueve.
- El sistema está en deadlock si en el grafo hay ciclos.

Detección + Recuperación

Detectar estado de Deadlock



Grafo de espera sin ciclos



Grafo de espera con ciclos
⇒ **deadlock**

Detección + Recuperación

Recuperar (romper el deadlock):

1. **Seleccionar una víctima:** dado un conjunto de transacciones en deadlock, determinar cuál/es transacciones deben retroceder.
2. **Retroceso:** determinar si una transacción debe ser retrocedida por completo o parcialmente (esto requiere información adicional del sistema).
3. **Inanición:** el sistema debe elegir como víctima a una transacción T_i un número finito de veces para evitar que entre en estado de inanición.

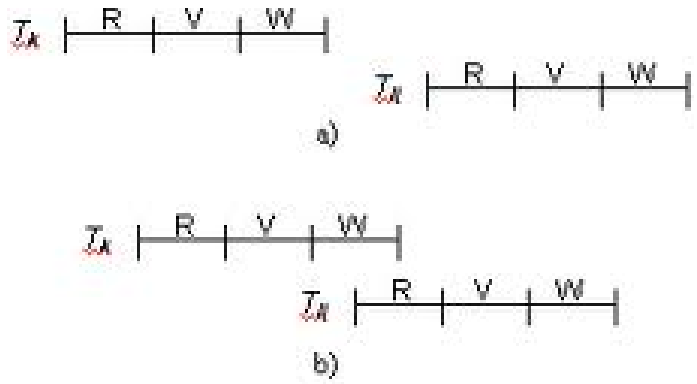
Inanición

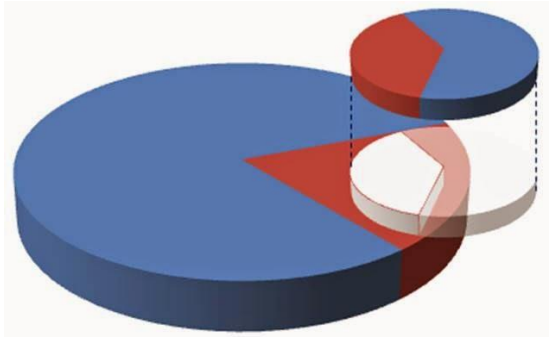
Inanición (starvation) – define la condición por la que una transacción es demorada indefinidamente a acceder a el/los recursos que requiere porque siempre se le da preferencia a otras.

- Sea un recurso Q, que es requerido con frecuencia por varias transacciones, y una transacción T_i que requiere acceso a dicho.
- Con protocolos de bloqueo, no consigue el permiso requerido (aunque no exista situación de interbloqueo).
- Entra en situación de deadlock y es elegida como víctima.
- Con protocolos de estampilla, no cumple las condiciones de estampillas y es obligada a reingresar.

- La condición de inanición de un protocolo se complementa siguiendo alguna política para administrar inanición

Bloqueos de Granularidad Múltiple





Protocolo de bloqueo con distinta granularidad

La granularidad para “Q”

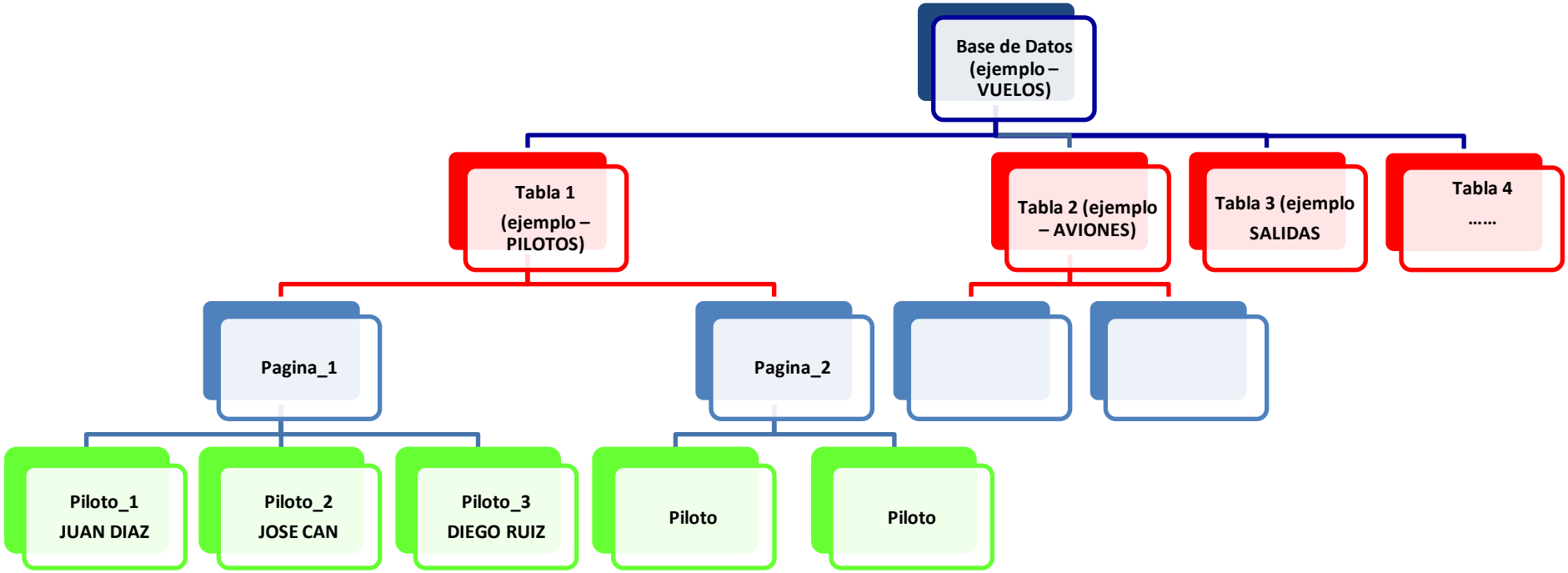
Granularidad de bloqueo (Lock-X/Lock-S (Q)) **–** cuál es la ‘*unidad*’ para los ítems de dato: *la base de datos, la tabla, una página o un registro*.

Depende!

- **Granularidad fina** (*hojas del árbol*): (+) alta concurrencia, (-) alta sobrecarga (overhead) por bloqueos.
- **Granularidad gruesa** (*niveles superiores*): (+) menor overhead por bloqueos, (-) concurrencia baja.

- Generalmente se usa granularidad fina (de tupla o registro).
- Los DBMS permiten al “*elegir*” la granularidad de bloqueo.

Jerarquía


Base de Datos (ejemplo – VUELOS)
Tabla 1 (ejemplo – PILOTOS)
Tabla 2 (ejemplo – AVIONES)
Tabla 3 (ejemplo SALIDAS
Tabla 4 ……
Pagina_1
Pagina_2
Piloto_1 JUAN DIAZ
Piloto_2 JOSE CAN
Piloto_3 DIEGO RUIZ
Piloto
Piloto

Niveles de Granularidad

- **(+)Ventajas de bloqueo a nivel de registro**:
  - Menor posibilidad de conflictos si las transacciones acceden a filas distintas.
  - Se puede mantener un lock por más tiempo sin demasiado riesgo de deadlock.
- **(-) Desventajas de bloqueo a nivel de registro**:
  - La tabla de bloqueos requiere más memoria.
  - Es más lento si una operación requiere de varios bloqueos, por ejemplo GROUP BY, se debe conseguir el lock de cada registro
- **Bloqueos a mayor nivel (*tabla* o *página*) son apropiados si:**
  - La mayoría de las operaciones son de lectura.
  - Muchos accesos usando consultas GROUP BY.

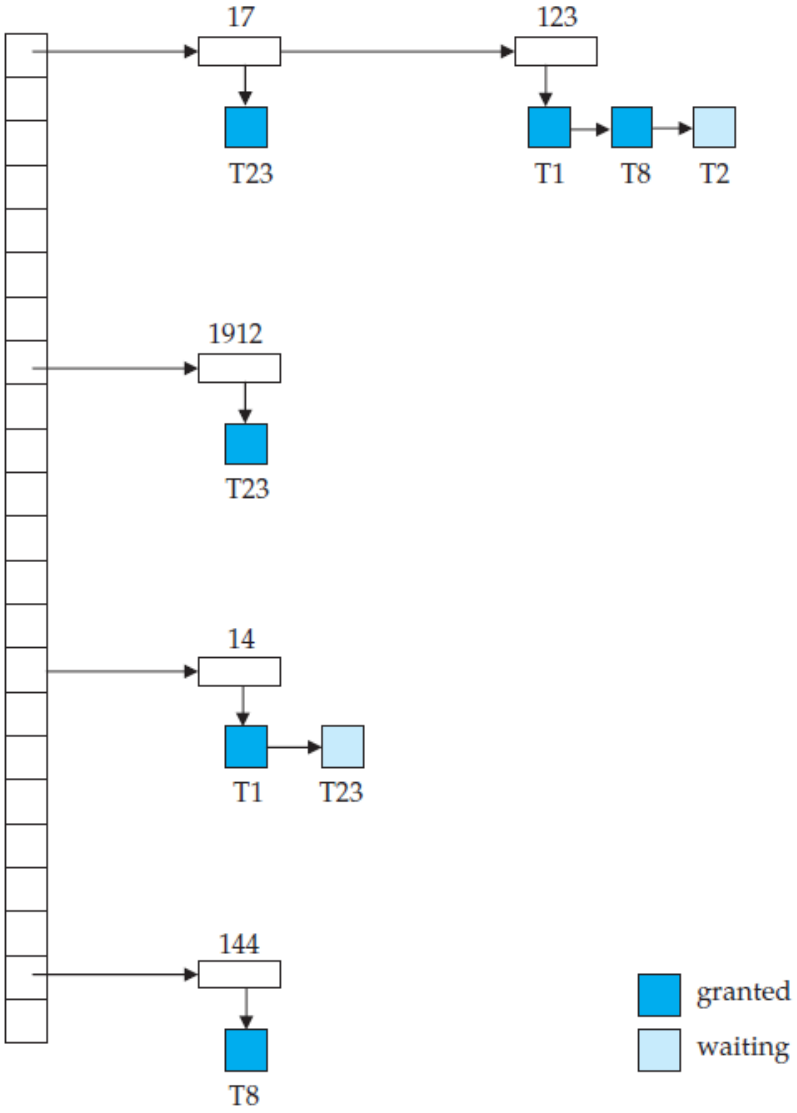


Figure 15.10 Lock table.

Tabla de Bloqueos

Lectura

- EBD_19_2015 Lock_Manager Silberschatz - Database System Concepts.pdf

- ¿Y con granularidades distintas?

Protocolo con Granularidad Múltiple

Bloqueos con granularidad múltiple – protocole que permite que coexistan bloqueos de datos en unidades distintas para diversas transacciones, esto es, una transacción decide individualmente el nodo del árbol para el que requiere un lock.

Ejemplo:

– Una transacción T_1 requiere un lock-s sobre la relación (o tabla) **CLIENTES**, mientras que otra transacción T_2 requiere un lock-X a nivel de registro para el cliente ID= 00123.

- El modo de lock (hasta ahora *exclusivo* o *compartido*) alcanza al nodo de la jeraquía y sus descendientes, si existen.
- Para hacer práctica la gestión de bloqueos con granularidad múltiple, se emplean tipos adicionales de bloqueo, llamados *locks de intención* (intention locks).

El gestor de bloqueos

- Algunos posibles niveles para un lock: base de datos, tabla, página o bloque, registro
- Tener un lock en un nivel de la jerarquía implica tener un lock del mismo modo sobre los descendientes.
  - Una transacción T_1 que tiene un lock-s sobre la relación (o tabla) **CLIENTES**, implícitamente también tiene un lock-s sobre cada registro de la tabla.
- **Problema!** el Gestor de Bloqueos debe asegurar permisos consistentes, aún cuando existen *locks implícitos:*
- Ejemplo:
  - T_1 tiene lock-s sobre el registro r_{123} de la tabla R; T_2 tiene lock-x sobre r_{425} de la tabla R; T_3 requiere lock-s sobre la tabla R.

¿cómo el gestor de bloqueos decide si se puede concederlo?

Intensión de Bloqueos

Solución

- Introducir nuevos modos: modo lock con intensión (*intention lock mode*)
  - *IS (intention-share mode):* en algún nivel inferior existen bloqueos explícitos compartidos.
  - *IX (intention-exclusive mode):* en algún nivel inferior existen bloqueos explícitos exclusivos o compartidos.
  - *SIX (shared, intention-exclusive mode):* el subárbol está bloqueado explícitamente en modo compartido y algún hijo en modo exclusivo.
- El problema del Gestor de Bloqueos es cómo asegurar permisos consistentes cuando existen *bloqueos implícitos:*

¿cómo determinar si se puede conceder o no un lock?

Matriz de Compatibilidades

- Las compatibilidades para todas las posibilidades de locks es :

pedido \ Y existe	IS	IX	S	S IX	X
IS	✓	✓	✓	✓	×
IX	✓	✓	×	×	×
S	✓	×	✓	×	×
S IX	✓	×	×	×	×
X	×	×	×	×	×

Protocolo de Bloqueo con Granularidad Múltiple

Para que T_i consiga bloquear un nodo Q del árbol:

- Respetar la matriz de compatibilidad.
- Bloquear primero la raíz en algún modo (puede ser en cualquier modo).
- Para obtener un **lock sobre un nodo Q en modo S o IS**, actualmente debe tener el lock sobre al padre de Q en IS o IX.
- Para obtener un **lock sobre un nodo Q en modo X, IX o SIX** actualmente debe tener el lock sobre el padre de Q en IX o SIX.
- T_i no se puede conseguir un nuevo lock si ya se libero alguno (dos fases).
- T_i solo pude liberar un lock si no mantiente locks sobre los hijos.

Ejemplos

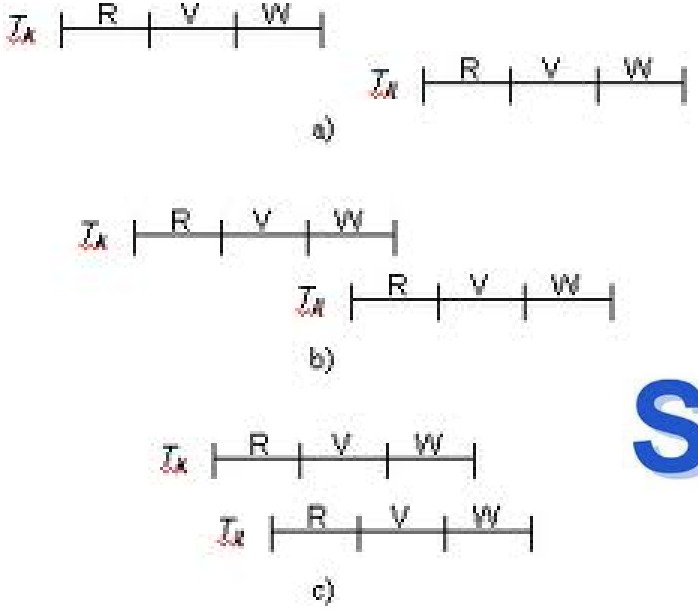
- T1 necesita leer y mostrar el domicilio del piloto Diego Ruiz (Lock-s a nivel de registro).
  - T1 debe conseguir IS sobre la base de datos, la tabla PILOTOS, la página_1 de Pilotos y S sobre el registro de Diego Ruiz.
- T2 actualiza las horas de vuelo del piloto Juan Roldán (requiere Lock-X a nivel de registro).
  - T2 necesita IX sobre la base de datos, la tabla PILOTOS, la página_1 de Pilotos y X sobre el registro de Juan Roldán.
- T3 consulta las horas de vuelo de todos los pilotos y actualiza las horas de vuelo de los que viajaron la última semana (requiere lock-S-IX a nivel de tabla.
  - T3 necesita IX sobre la base de datos y SIX sobre la tabla pilotos y X sobre los registros pilotos que cumplen con la condición.

Protocolo Granularidad-Multiple

Propiedades del Protocolo

- El protocolo de bloqueos de granularidad múltiple con dos fases garantiza *planificaciones serializables*.
- *No está libre de deadlock*.

Implementaciones en Control de Concurrencia



Multiversión + 2PL

SQL

Multiversión + 2PL

- Diferencia entre **transacciones sólo de lectura (TrR)** y **transacciones de actualización (TrW)** .
- Las **TrW** deben adquirir los lock (S ó X) y retenerlos hasta que la transacción termine (dos fases)
  - Cada escritura exitosa crea una nueva versión del ítem de datos.
  - Cada versión de un ítem de dato tiene una estampilla de tiempo cuyo valor se obtiene de un **ts-counter** que se incrementa con el proceso de commit de la transacción.
- A las **TrR** se les asigna una estampilla de tiempo que corresponde con el valor actual de **ts-counter** antes de iniciar su ejecución. Las lecturas siguen el protocolo multiversión

Multiversión + 2PL

- Si TrW_i requiere read(Q), debe conseguir un lock-S(Q) sobre la última versión de Q.
- Si TrW_i requiere un write(Q), debe obtener un lock-X(Q) sobre la última versión de Q y crear la nueva versión con estampilla de tiempo infinito.
- Cuando TrW_i se completa, el proceso de commit:
  - Asigna a TrW_i la estampilla de tiempo ts-counter + 1 y guarda el nuevo valor de ts-counter.
- Las TrR_j que comiencen después de que TrW_i cometió verán el valor actualizado.
- Las TrR_j que comiencen antes de que TrW_i cometan verán el valor anterior.

Análisis de MV2PL

Ventajas

- Genera planificaciones serializables.
- No tiene problemas de retrocesos en cascada ni planificaciones no recuperables.

Desventajas.

- Requiere mantener múltiples tuplas.
- Debe considerar gestión de garbage collected.
- La transacción debe indicar su tipo (lectura o escritura)

Niveles de aislamiento más débiles

SQL estándar admite ejecuciones concurrentemente con niveles de aislamiento inferiores a serializable

Definiciones Previas

Dirty Reads (*lecturas sucias*) – cuando se permite a una transacción *leer datos de transacciones no cometidas y hasta cometer* antes que la transacción que generó los datos leídos.

- **T1 = write(x); ….** **T1** *abort*
- **T2 =** **read(x); y=x; write(y);** ***commit***

- Dirty-read causa planificaciones no recuperables y retrocesos en cascada.

Dirty Writes (*escrituras sucias*) – si se autoriza a una transacción *escribir* datos ya escritos por una transacción no cometida:

- **T1 = write(x); ….** ***abort***;
- **T2 =** **write(x); ….** ***write(Y);***

Definiciones Previas

- Las operaciones **INSERT**, **DELETE** y **UPDATE** de SQL son de escritura y requieren de un *lock-X*.
- Si el lock es a nivel de tupla (granularidad fina), se podrían generar planificaciones no serializables o de "phantom":
  - T_1: **SELECT** SUM(saldo) **FROM** cuentas **WHERE** cli_nombre='María'…
  - T_2: **INSERT INTO** cuentas **VALUES** (10, 'María',100$).; ….
  - T_1: **SELECT** COUNT(*) **FROM** cuentas **WHERE** cli_nombre='María'…

Phantom – T_2 inserta la nueva tupla en *cuentas* sin ser afectada por los locks actuales de T_1. T_1 cree que tiene lock-x sobre todas las tuplas que satisfacen el predicado y pero eso dejó de ser verdad!

Definiciones Previas

Repeatable Read – una transacción T_1 lee un valor Q, concurrentemente otra transacción T_2 lo modifica. Se define de *repeateable read* cuando se asegura que T_1 siga viendo el mismo valor de Q.

Contraejemplo:

- Supongamos la siguiente secuencia de ejecución:
  - **T_1**: <*start*, T_1> **SELECT** saldo **FROM** cuentas **WHERE** cli_nombre='María'.
  - **T_2:** <*start*, T_2> **UPDATE** cuentas **SET** saldo= saldo +10 **WHERE** cli_nombre= 'María' <*commit*, T_2>. <- *debe esperar*
  - **T_1**: **SELECT SUM**(saldo) **FROM** cuentas **WHERE** cli_nombre='María' <*commit*, T_1>.
- Con repeteable reade se espera que la segunda lectura de T_1 no vea los cambios de T_2.

Niveles de Aislamiento de SQL'92

- SQL'92 define cuatro niveles de aislamiento. Los menos estrictos pueden generar ejecuciones no serializables:

  Serializable

  Repeatable read – autoriza la lectura de registros que pertenecen a transacciones cometidas. Si la misma transacción necesita leer el mismo registro varias veces lee el mismo valor.

  Read committed – autoriza el la lectura de registros que pertenecen a transacciones cometidas.

  Read uncommitted – permite leer datos de generados por transacciones no cometidas.

Niveles de Aislamiento

Nivel	Dirty Reads	NonRepeatable Reads	Phantom
READ UNCOMMITTED	✓	✓	✓
READ COMMITTED	–	✓	✓
REPEATABLE READ	–	–	✓
SERIALIZABLE	–	–	–

Niveles de Aislamiento

- Los DBMS definen un nivel de consistencia por defecto, generalmente *read committed o repeteable read.*
- Es posible explícitamente cambiarlo:
  - Modificar los parámetros del sistema.
  - Modificar las condiciones para una transacción en particular con la instrucción SQL
    - **SET ISOLATION LEVEL …**

Video

- https://www.youtube.com/watch?v=kTU3PtP_pkg

DBMS Relacionales

Los DBMS para asegurar serializabilidad requieren:

1. Bloquear tablas completas, o con granularidad más fina bloquear índices.
2. Exigir conservar los bloqueos por un tiempo mayor al necesario (hasta el final), resultando en una performance inadecuada para ciertas aplicaciones.

- **SQL'92 define niveles de aislamiento menos estrictos** (*weaker isolation levels*) que permiten mejorar la concurrencia.
- Según cada problema particular, los desarrolladores pueden decidir cuál nivel de aislamiento a usar.

Bloqueos a Nivel de Índice

Index locking protocols – provee concurrencia y previene *phantom* fijando los locks en el registro índice, en lugar de sobre los datos.

- Todas las relaciones o tablas deben tener por los menos un índice y el acceso a las tuplas debe ser a través del índice, que se bloquea con lock-S
- Una transacción que inserta una tupla requiere escribir el índice lock-X.
- Puede prevenir el fenómeno *phantom*.

Temas de la clase de hoy

- Protocolos de control de concurrencia:
  - Granularidad Múltiple.
  - Multiversión y 2PL
- Gestión de deadlocks
  - Prevención.
  - Detección
- **Bibliografía**
  - “Database System Concepts” – A. Silberschatz. Capítulo 16 y 26 a 29.