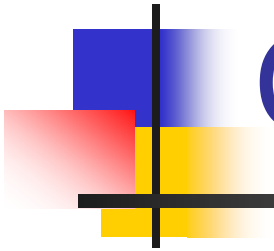
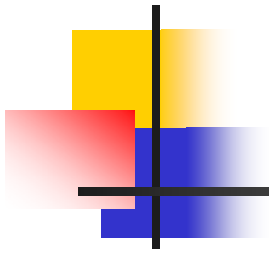


Transacciones, Recuperación y Control de Concurrencia

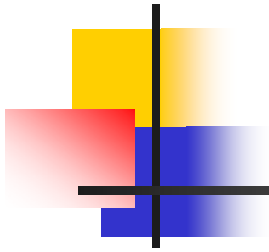
Diseño de Bases de Datos Relacionales
Curso 2011/2012

Sergio Ilarri
silarri@unizar.es



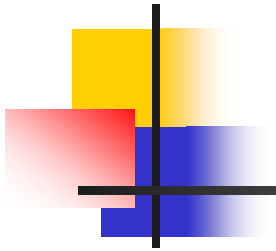
Transacciones (I)

- Transacción
  - Secuencia de operaciones que deben ejecutarse de forma atómica (unidad lógica)
  - La operación más simple con verificación de consistencia
- Ejemplo motivador: transferencia bancaria
- Deben definirse las transacciones necesarias de forma adecuada (tan pequeñas como sea posible pero asegurando la integridad)



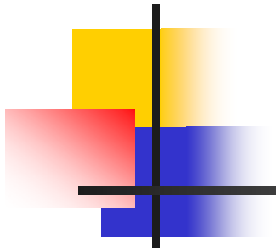
Transacciones (II)

- Ciclo de vida de una transacción
  - Inicio
  - Lecturas/escrituras de elementos de la BD
  - Final (pueden necesitarse verificaciones)
  - Confirmación (COMMIT)
  - Anulación (ROLLBACK)
- Control de concurrencia
  - Sistema multi-usuario
- Fallo de una transacción → recuperación



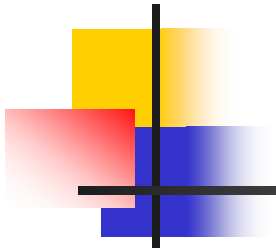
Transacciones (II)

- Propiedades ACID:
  - **Atomicidad** (*atomicity*) → método de recuperación
    - Se ejecuta todo (*commit*) o nada (*rollback*)
  - **Consistencia** (*consistency*) → programador y/o SGBD (restricciones de integridad)
    - Se pasa de un estado consistente a otro
  - **aIslamiento** (*isolation*) → método de control de concurrencia y a veces también el programador (ej., bloqueo en 2 fases)
    - No interfiere con otras, no lee resultados intermedios de transacciones no terminadas
  - **Durabilidad** (*durability*) → método de recuperación
    - En caso de éxito (*commit*), los cambios son persistentes, incluso si hay fallos de otras



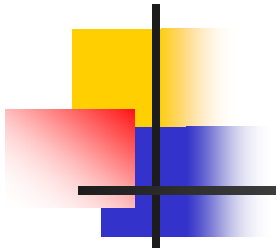
Problemas de concurrencia (I)

- La ejecución concurrente de transacciones de forma incontrolada puede dar lugar a problemas:
  - Problema de la actualización perdida
  - Problema de la lectura sucia
    - También: problema de la actualización temporal, problema de la lectura de datos no confirmados
  - Problema del resumen incorrecto (resultados incoherentes)
  - Problema de la lectura no repetible
    - También: problema de valores contradictorios de los datos
- Para evitarlos, el control de la concurrencia es necesario



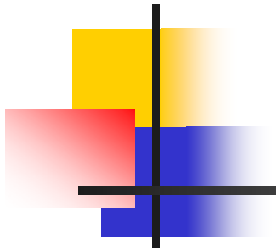
Problemas de concurrencia (II)

- Problema de la actualización perdida
  - T1 lee las existencias disponibles de un producto (E)
  - T2 también lee las existencias disponibles (E)
  - T1 incrementa dichas existencias en 100 unidades (E+100)
  - T2 también incrementa dichas existencias en 100 unidades (E+100)
  - El resultado final (E+100) sería incorrecto (debería ser E+200), ya que se ha perdido la actualización de T1
  - Ambas transacciones leen el valor antes de que lo cambie la otra…



Problemas de concurrencia (III)

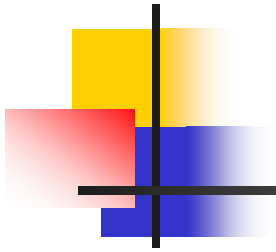
- Problema de la lectura sucia
- T1 lee las existencias disponibles de un producto (E)
- T1 incrementa dichas existencias en 100 unidades (E+100)
- T2 lee las existencias disponibles de dicho producto (E+100)
- T1 aborta, debido a un error, deshaciéndose sus cambios (las existencias pasan de nuevo a valer E)
- T2 incrementa las existencias en 100 unidades (E+200)
- El resultado final (E+200) sería incorrecto (debería ser E+100) ya que T1 ha abortado su actualización
- Una transacción actualiza un ítem que luego lee otra. Después falla (por tanto, la segunda lee un dato que "nunca existió")



Problemas de concurrencia (IV)

- Problema del resumen incorrecto
- T1: cálculo del saldo total de todas las cuentas de un banco
- T2: transferencia de una cuenta bancaria a otra (c1 → c2)
- T2 sustrae X del saldo de c1
- T1 lee el saldo de c1
- T1 lee el saldo de c2
- T2 añade X al saldo de c2
- El saldo total calculado por T1 será menor que el real por X
- Una transacción calcula un agregado de registros mientras otra actualiza algunos de esos registros → algunos valores se consideran antes de la actualización y otros después

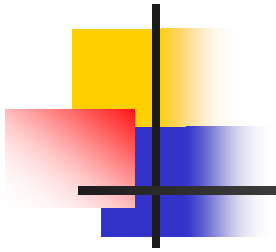
Problemas de concurrencia (V)



- Problema de la lectura no repetible
  - T1 consulta asientos disponibles en varios vuelos:
    - Hay N asientos disponibles en IB245 y N' en FJ143
  - T2 reserva X asientos en el vuelo IB245
  - T1 decide finalmente reservar en el vuelo IB245
    - Al volver a leer los asientos disponibles en dicho vuelo, se obtiene N-X (no coincide con los de antes)
    - Incluso podría no haber ya asientos disponibles…
    - Una transacción lee dos valores diferentes para un ítem, debido a una modificación intermedia por parte de otra
- Problema de la lectura fantasma (tuplas que no existían antes)

Planes serializables

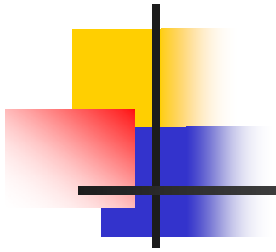
- Dadas las transacciones T1, T2, ... Tn,
  - T1 → operaciones O_{11},O_{12},..$O_{1\ m1}$
  - T2 → operaciones O_{21},O_{22},..$O_{2\ m2}$
  - …
  - Tn → operaciones O_{n1}, O_{n2}..$O_{n\ mn}$
- Plan de ejecución:
  - Ej.: O_{11}, O_{21}, O_{n1}, O_{n2}, O_{12}, O_{22}, …, $O_{1\ m1}$, $O_{2\ m2}$, …, $O_{n\ mn}$
  - Una intercalación de todas las operaciones O_{ij} donde para todo i, O_{i1} se ejecuta antes que O_{i2} ... $O_{i\ mi}$
- Plan serializable → resultado igual al producido por alguno de los posibles planes seriales
  - Ej.: operaciones de T2, de T1, de Tn, …., de T3



Serializabilidad

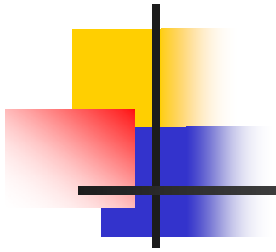
- Utilizada como criterio de corrección
- Determinar si un determinado plan es serializable es un problema NP-completo
- Idea: imponer restricciones a la libre intercalación de operaciones de transacciones para garantizar que el plan resultante es serializable

Técnicas de control de concurrencia (I)

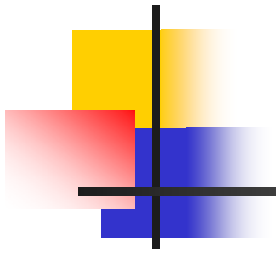


- Objetivo: evitar interferencias entre transacciones → aislamiento
- Solución trivial: cada transacción se ejecuta en exclusión mutua
  - Muy restrictivo
  - Evita la concurrencia y las BD están pensadas para acceso multi-usuario (múltiples aplicaciones)
  - En lugar de eso, intercalar acciones para que el resultado sea como en exclusión mutua...
    - Aquí es donde el concepto de serializabilidad juega un papel fundamental

Técnicas de control de concurrencia (II)

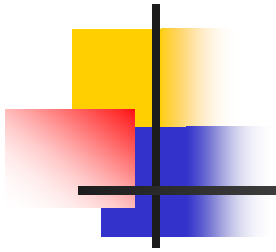


- Principales técnicas utilizadas:
  - Técnicas pesimistas
  - Técnicas optimistas
  - Técnicas de control de concurrencia multi-versión



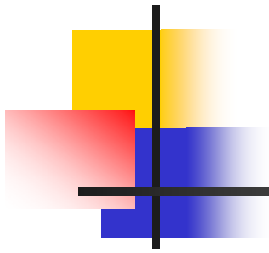
Técnicas pesimistas

- Técnicas pesimistas:
  - Técnicas de bloqueo (*locks*)
  - Técnicas de marcas de tiempo (*time-stamping*)
- Se impiden ciertas operaciones si son sospechosas de producir planes no serializables. Control de las transacciones durante la ejecución (no sólo al final)
- Utilizan protocolos (reglas) para garantizar la serializabilidad de los planes de ejecución (*schedules*)



Técnicas de bloqueo (I)

- Utilizadas en la mayoría de los SGBD comerciales
- Idea: bloquear ítems de datos para evitar su acceso concurrente desde múltiples transacciones (sincronizar el acceso)
- Cerrojo (*lock*):
  - Variable asociada a un ítem de datos, que describe su estado con respecto a las operaciones permitidas
  - Riesgos: bloqueo mortal (*deadlock*), inanición (*starvation*)



Técnicas de bloqueo (II)

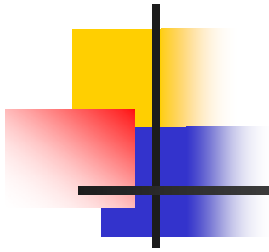
- Cerrojos binarios:
  - 0 ó 1 (accesible o no accesible)
  - Operaciones *lock()* y *unlock()* → indivisibles (exclusión mutua, región crítica, cola de espera)
  - Simples pero restrictivos → no usados
- Cerrojos multi-modo (3 posibles estados):
  - *Shared locks* (*S locks*, *read locks*)
  - *Exclusive locks* (*X locks*, *write locks*)
  - Operaciones *read_lock()*, *write_lock()*, *unlock()* → indivisibles
  - Permiten accesos simultáneos en modo lectura

Técnicas de bloqueo (III)

- Granularidad de los ítems de datos X:
  - Un atributo
  - Una tupla
  - Una tabla
  - Un bloque de disco
  - Un fichero entero
  - La BD entera
  - …

  Operaciones básicas de acceso a la BD: read(X), write(X)

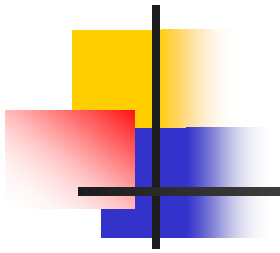
- Sobrecarga de adquisición y gestión de cerrojos vs. menor nivel de concurrencia si la granularidad de los ítems de datos es mayor



Técnicas de bloqueo (IV)

- Los cerrojos no garantizan la serializabilidad por sí solos
  - Hace falta un protocolo adicional referente al posicionamiento de las operaciones de bloqueo y desbloqueo dentro de una transacción
  - Protocolo más utilizado: protocolo de bloqueo en 2 fases
    - ¡No confundir con el protocolo de *commit* en 2 fases! (bases de datos distribuidas)

Protocolo de bloqueo en 2 fases (2PL)

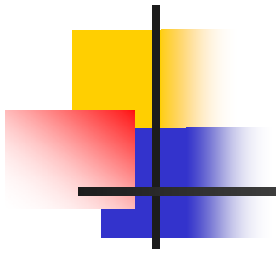


- Idea: no hacer ningún *lock()* después de algún *unlock()*
- 2 fases:
  - Fase de crecimiento/expansión: solicitud de *locks*
  - Fase de devolución: realización de *unlocks*
- Garantiza la serializabilidad → evita la necesidad de comprobarla
  - Pero puede limitar la concurrencia (restrictivo) → impide algunos planes serializables
- Variante más popular: 2PL estricto (planes estrictos)
  - No se libera ningún cerrojo exclusivo (de escritura) hasta que se finaliza
    - → ninguna transacción podrá leer o escribir ese ítem → evita *rollback* en cascada
    - → permite una fácil recuperación (restaurar valores viejos antes del write)

Bloqueos mortales (I)

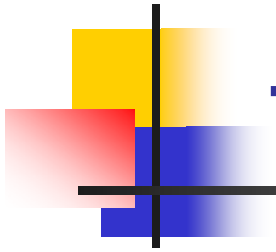
- Prevención
  - Cada transacción obtiene todos los cerrojos al principio: o todos o ninguno (2PL conservador): problema de *livelock*
  - Los elementos están ordenados y los cerrojos hay que obtenerlos en orden (responsabilidad del programador)
  - Ordenación de transacciones por marcas de tiempo
    - *Wait-die* → si TS(Ti) < TS(Tj), entonces Ti puede esperar; de otro modo, Ti muere. Transacciones + viejas esperan por otras + jóvenes.
    - *Wound-wait* → si TS(Ti) < TS(Tj), entonces Ti hiere a Tj (que aborta); de otro modo, Ti puede esperar.
      Transacciones + jóvenes esperan por otras + viejas.
    - Las transacciones se reinician con la misma marca de tiempo

    Ciclos no posibles
  - Otras: abortar si no es posible obtener un cerrojo, etc.



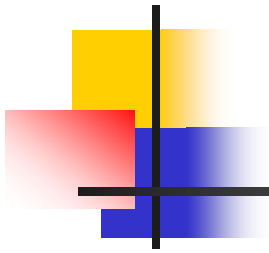
Bloqueos mortales (II)

- Detección y recuperación
  - Grafo de esperas → si hay un ciclo abortar a una
    - Problema de *livelock*
    - ¿Cuándo se comprueba? → #transacciones, tiempo de espera por cerrojos, periódicamente, etc.
    - Selección de la víctima
      - La transacción que lleva menos tiempo ejecutándose
      - La transacción que está bloqueando un mayor número de transacciones
      - La transacción que menos veces se ha abortado
      - …
  - Uso de *timeouts*



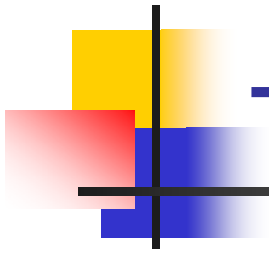
Técnicas de marcas de tiempo (I)

- Marca de tiempo (*timestamp*)
  - Identificador único de cada transacción
  - Generado automáticamente por el sistema
  - A cada elemento X de la BD se le asigna también la marca de tiempo de la transacción más nueva (mayor marca de tiempo) que lo ha leído y escrito:
    - TS_lect(X) y TS_escr(X)
  - Uso del orden de las marcas de tiempo para garantizar la serializabilidad
    - Pero puede haber planes serializables que los impida...



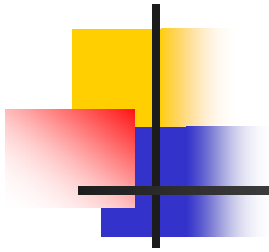
Técnicas de marcas de tiempo (II)

- Si una transacción T quiere escribir en X
  - Si TS_lect(X) > TS(T) entonces abortar
    - Una transacción más nueva ya ha leído X antes de que T tuviera la oportunidad de modificarlo
  - Si TS_escr(X) > TS(T) entonces no escribir y seguir (regla de escritura de Thomas)
    - Una transacción más nueva ya ha escrito el valor de X, por lo que se puede ignorar la escritura por parte de T
    - Problema potencial: la transacción más nueva puede abortar… → seguir la pista a las escrituras tentativas y a sus valores previos
  - En otro caso escribir y TS_escr(X):=TS(T)



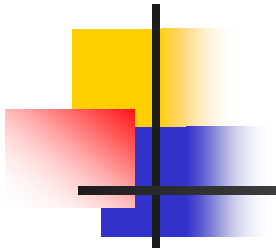
Técnicas de marcas de tiempo (III)

- Si una transacción T quiere leer de X
  - Si TS_escr(X) > TS(T) entonces abortar
    - Una transacción más nueva escribió X antes de que T tuviera la oportunidad de leer su valor
  - Si TS_escr(X) <= TS(T) entonces leer de X y TS_lect(X):=máximo(TS(T), TS_lect(X))



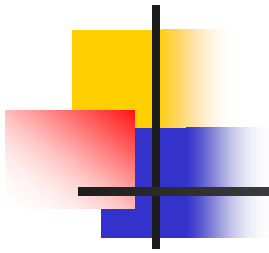
Técnicas optimistas (I)

- También llamadas técnicas de validación o técnicas de certificación
- No imponen restricciones, no requieren bloqueo, se comprueban posibles interferencias al final
- 3 fases:
  - Fase de lectura
  - Fase de validación (¿ha habido algún conflicto? ¿alguna actualización violaría la serializabilidad?)
  - Fase de escritura (si se valida satisfactoriamente; de otro modo, abortar, dado que puede haber habido interferencias)
- Las actualizaciones se realizan inicialmente sobre copias locales (≈ versiones)



Técnicas optimistas (II)

- Adecuadas cuando hay pocas interferencias entre transacciones (optimismo); si no, coste de reiniciar
- Vamos a ver una técnica concreta
- Esta técnica utiliza:
  - Marcas de tiempo de las transacciones, instantes de inicio y fin de algunas fases
  - Conjuntos de escritura y lectura de las transacciones
- En la fase de validación de una transacción, se comprueba que esta no interfiere con ninguna otra transacción confirmada ni con otras actualmente en su fase de validación



Técnicas optimistas (III)

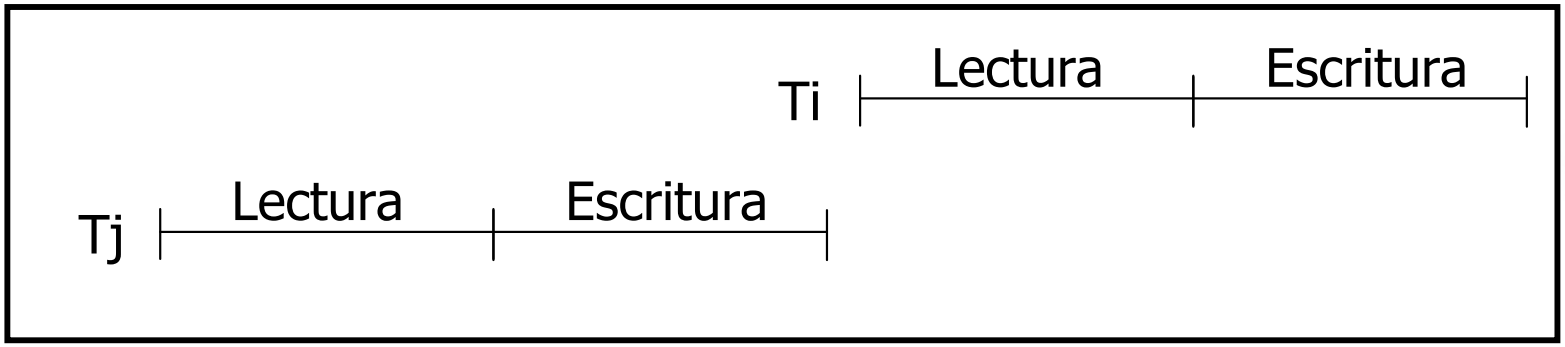
- Fase de validación para Ti
  - Antes de hacer *commit* se comprueba si hay interferencias
  - Para cada Tj confirmada o actualmente en su fase de validación, Ti y Tj no tienen interferencias si:
    1. Tj ha terminado su fase de escritura antes de que Ti haya comenzado su fase de lectura
    2. Ti comienza su fase de escritura después de que Tj complete su fase de escritura y Ti no lee elementos escritos por Tj
    3. No hay elementos en común entre los que lee y escribe Ti con los que escribe Tj, y Tj ha terminado su fase de lectura antes de que Ti termine su fase de lectura

Si, para cada Tj, se cumple cualquiera de estas 3 condiciones no hay interferencia. Se comprueban en orden (dado que están ordenadas por complejidad de su evaluación)

Técnicas optimistas (IV)

Fase de validación:

1. Tj ha terminado su fase de escritura antes de que Ti haya comenzado su fase de lectura

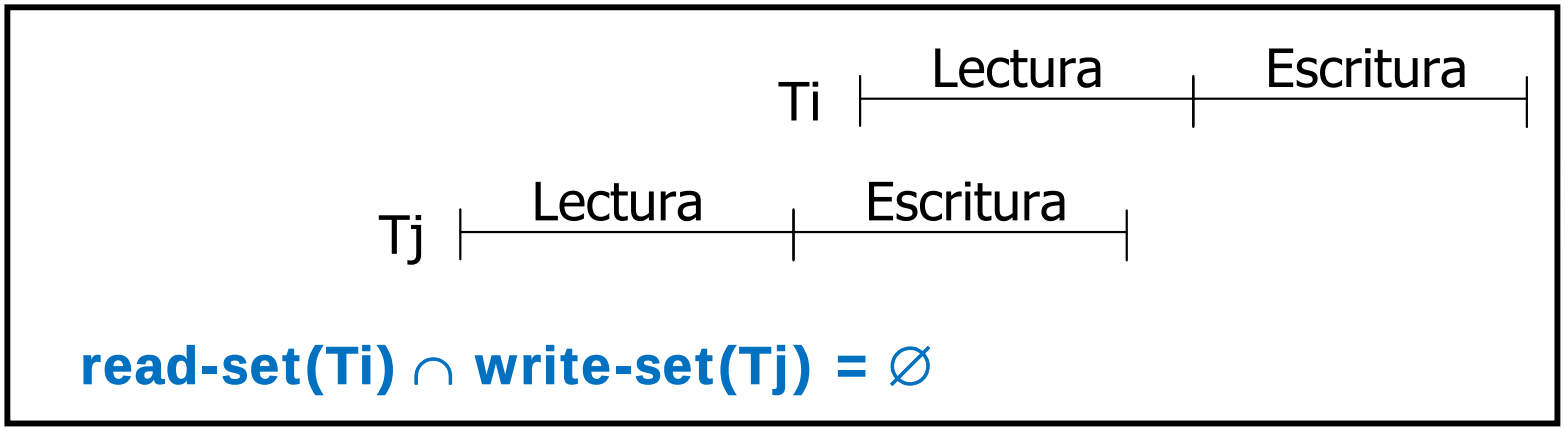


En este caso tenemos una ejecución en serie

Técnicas optimistas (V)

Fase de validación:

2. Ti comienza su fase de escritura después de que Tj complete su fase de escritura y Ti no lee elementos escritos por Tj

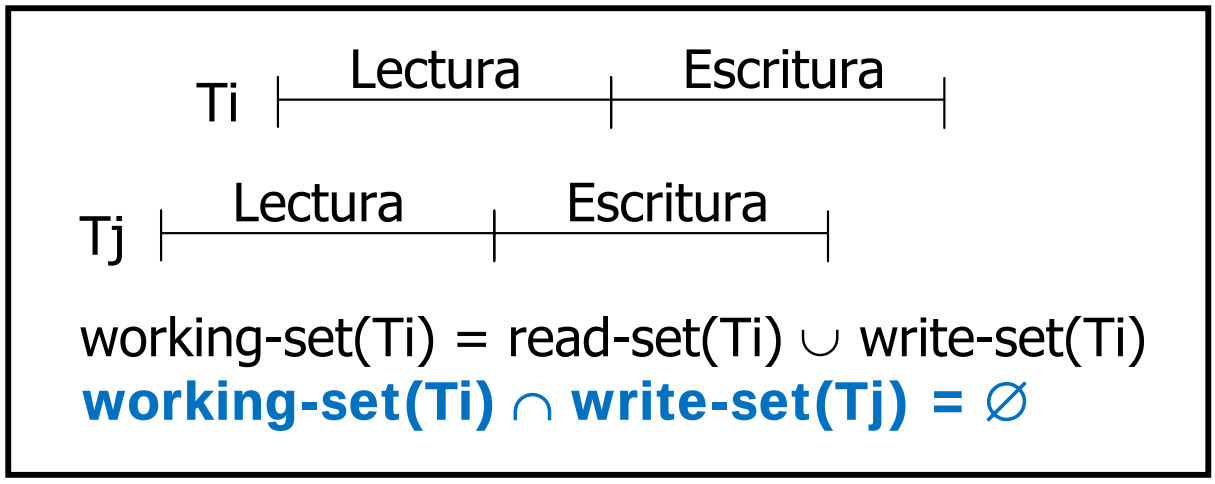


-Las escrituras de Tj no afectan a las lecturas de Ti (por la condición en azul).
-Tj termina de escribir antes de que empiece Ti → no puede sobreescribir cambios de Ti
-Ti no puede afectar a la fase de lectura de Tj

Técnicas optimistas (VI)

Fase de validación:

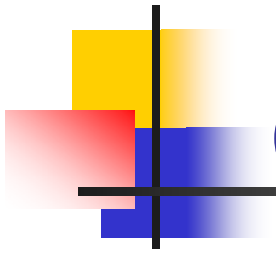
3. No hay elementos en común entre los que lee y escribe Ti con los que escribe Tj, y Tj ha terminado su fase de lectura antes de que Ti termine su fase de lectura



working-set(Ti) = read-set(Ti) ∪ write-set(Ti)
working-set(Ti) ∩ write-set(Tj) = ∅

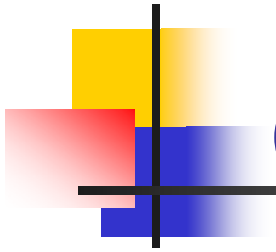
-Lo que escribe Tj no influye en lo que pueda leer Ti (debido a la condición en azul)
-Todas las lecturas de Tj se hacen antes de que Ti empiece a escribir

Técnicas de control de concurrencia multi-versión (I)

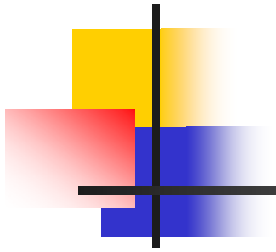


- Utilizan múltiples versiones de los ítems de datos
- Objetivo: permitir lecturas que en otras circunstancias podrían lleva a la transacción que lee a abortar
- Marca de tiempo para escrituras:
  - Se genera una nueva versión cada vez que se escribe un objeto
  - Su marca de tiempo de escritura es la marca de tiempo de la transacción correspondiente
  - Cada transacción T lee la versión de un ítem que es apropiada para ella, que es la versión escrita inmediatamente antes de la ejecución teórica de T (marca temporal de T)

Técnicas de control de concurrencia multi-versión (II)



- Marca de tiempo para lecturas:
  - Cada versión tiene una marca de tiempo que indica la marca de tiempo de la última transacción que la leyó (la marca de tiempo de transacción más alta que la leyó)
  - Si una transacción intenta realizar una escritura sobre un ítem pero la marca de tiempo de dicha escritura sería inferior a la marca de tiempo de lectura de la versión previa, entonces se aborta la escritura
- Si una versión tiene un tiempo de escritura tal que no existe ninguna transacción activa con marca temporal menor, entonces todas las versiones anteriores a dicha versión pueden eliminarse

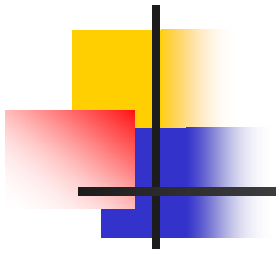


Recuperación (I)

- Tipos de fallos:
  - Fallo informático (*system crash*): hardware, software, red
  - Error del sistema o transacción: desbordamiento de un entero, división por 0, errores de parámetros, errores de programación, etc.
  - Errores locales o condiciones de excepción detectadas por una transacción: datos que no se encuentran, saldo insuficiente, etc.
  - Control de concurrencia: transacciones abortadas para garantizar la serializabilidad o para romper un abrazo mortal
  - Fallo del disco (lectura/escritura)
  - Problemas físicos y catástrofes: fuego, inundación, robo, sabotaje, fallos humanos (sobreescritura de datos incorrecta, borrado de datos, …), etc.

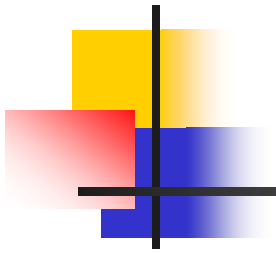
Recuperación (II)

- Fichero de *log* (bitácora) + backups
- Registros de *log* (caso general, *undo/redo logging*)
  - <**comienza-transacción**, numTrans>
  - <**escritura**, numTrans, idDato, valViejo, valNuevo>
  - <**lectura**, numTrans, idDato, val> → *Salvo si el protocolo evita rollbacks en cascada*
  - <**termina_transacción_con_éxito**, numTrans >
  - <**punto_comprobación**, numTrans, numPuntoComprob>
- Puntos de comprobación / salvaguarda (*checkpoints*)
- Se escribe en el fichero de *log* antes que en la BD
  - Debe almacenarse de forma persistente



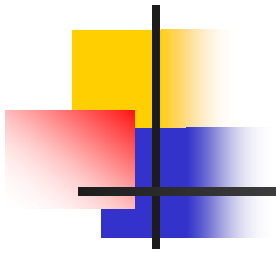
Recuperación (III)

- Tipos de registros:
  - *Undo logging*
    - *<escritura, numTrans, idDato, valViejo>*
    - Sólo deshace transacciones incompletas
    - Para hacer *commit*, se requiere antes escribir todos los datos cambiados en la BD en disco
  - *Redo logging*
    - *<escritura, numTrans, idDato, valNuevo>*
    - Sólo rehace cambios realizados por transacciones confirmadas
    - Es preciso actualizar el fichero de *log* incluyendo el *commit*, antes de modificar los datos en disco
    - Las transacciones incompletas se pueden tratar durante la recuperación como si nunca hubieran existido



Recuperación (IV)

- Tipos de registros (II):
  - *Undo/redo logging*
    - <escritura, numTrans, idDato, valViejo, valNuevo>
    - Más flexibilidad en cuanto al orden relativo con el que los registros de commit de *log* y los cambios en la BD se escriben en el disco
      - Antes de cambiar un dato en disco hay que escribir el cambio en el *log* en disco
    - Rehace las transacciones confirmadas (empezando por las más antiguas) y deshace las transacciones incompletas (empezando por las más nuevas)



Recuperación (V)

- ¿Puedo eliminar todos los registros del *log* para transacciones ya confirmadas?
  - No, podría haber un fallo catastrófico de disco… → recurrir a una copia de seguridad + *log* (*log online*, *log* archivado)

Concurrencia en Oracle (I)

validación de transacciones pendientes: **COMMIT;**

SET AUTOCOMMIT ON;
SET AUTOCOMMIT OFF;

establecer un pto. de "salvaguarda": **SAVEPOINT nombrePuntoSalvaguarda;**

deshacer transacciones pendientes: **ROLLBACK;**

ROLLBACK TO nombrePuntoSalvaguarda;

bloquear registros recuperados:

```
SELECT listaColumnas
FROM tablas
FOR UPDATE [OF atributosTablas];
```

```
CURSOR nombreCursor IS
  sentenciaSelect
  FOR UPDATE [of listaColumnas] [NOWAIT];
```

bloquear una tabla: **LOCK TABLE tabla IN modoBloqueo MODE;**

ROW SHARE, ROW EXCLUSIVE,
SHARE UPDATE, SHARE,
SHARE ROW EXCLUSIVE,
EXCLUSIVE

La desconexión finaliza una transacción, no hay un comenzar_transacción() explícito → ROLLBACK y COMMIT implícitos

Concurrencia en Oracle (II)

prohibición de modificaciones: **SET TRANSACTION READ ONLY**

situación por defecto: **SET TRANSACTION READ WRITE**

} Modo de acceso

Nivel de aislamiento: {

SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;

SET TRANSACTION ISOLATION LEVEL READ COMMIT;

Niveles de aislamiento: READ_COMMITTED, SERIALIZED

- *No hay unlocks explícitos en Oracle → implícitos al confirmar o abortar una transacción*

Comprobación diferida de restricciones (I)

```
CREATE TABLE Gallina (
   idGallina INT PRIMARY KEY,
   idHuevo  INT REFERENCES Huevo(idHuevo));

CREATE TABLE Huevo (
   idHuevo INT PRIMARY KEY,
   idGallina INT REFERENCES Gallina(idGallina));
```

Comprobación diferida de restricciones (II)

```
CREATE TABLE Gallina(idGallina INT PRIMARY KEY, idHuevo INT);
CREATE TABLE Huevo(idHuevo INT PRIMARY KEY, idGallina INT);

ALTER TABLE Gallina ADD CONSTRAINT fkRefHuevo
FOREIGN KEY(idHuevo) REFERENCES Huevo(idHuevo)
INITIALLY DEFERRED DEFERRABLE;

ALTER TABLE Huevo ADD CONSTRAINT fkRefGallina
FOREIGN KEY(idGallina) REFERENCES Gallina(idGallina)
INITIALLY DEFERRED DEFERRABLE;

INSERT INTO Gallina VALUES(4, 5);
INSERT INTO Huevo VALUES(5, 4);
COMMIT;
```

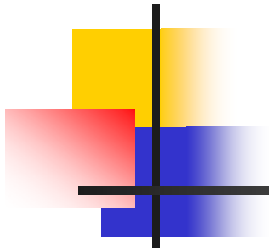
Comprobación diferida de restricciones (III)

```
ALTER TABLE Huevo DROP CONSTRAINT fkRefGallina;

ALTER TABLE Gallina DROP CONSTRAINT fkRefHuevo;

DROP TABLE Huevo;

DROP TABLE Gallina;
```

Niveles de Aislamiento (I)

- En algunos casos, podemos reducir la sobrecarga de la gestión de cerrojos
- SQL92 define 4 niveles de aislamiento:

Nivel	Lectura sucia	Lectura no repetible	Lectura fantasma
Read uncommitted	Posible	Posible	Posible
Read committed	No posible	Posible	Posible
Repeatable read	No posible	No posible	Posible
Serializable	No posible	No posible	No posible



Niveles de Aislamiento (II)

- Oracle ofrece 2 niveles de aislamiento:
  - *Read committed* (por defecto)
    - Evita el problema de la lectura sucia
    - Evita el problema de la actualización perdida
  - *Serializable*
- Además, ofrece un modo de sólo lectura
  (no parte del estándar)
  - *SET TRANSACTION READ ONLY*
  - Sólo se verán los cambios confirmados antes de que empezara la transacción