

# Transacciones, Recuperación y Control de Concurrencia

# Transacciones



| Transacción: colección de operaciones que forman una única unidad lógica de trabajo en una BD

| Control concurrencia

{ Sistemas multiusuario: ejecución intercalada

| Recuperación

{ Para cuando una transacción falla

| Vida de una transacción

{ Inicio

{ Lecturas/escrituras de elementos de la BD

{ Final (pueden hacer falta algunas verificaciones)

{ Confirmación (COMMIT) o anular (ROLLBACK)

# Transacciones



| Toda transacción debe cumplir el principio ACID

{ Atómica: se ejecuta todo (commit) o nada (rollback)

- Debe garantizarlo el método de recuperación del SGBD

{ Consistente: pasa de un estado consistente a otro

- Debe garantizarlo el programador y el SGBD (restr. int.)

{ aislada: no lee resultados intermedios de otras transacciones que no han terminado

- Debe garantizarlo el método de control de concurrencia y el programador (ej: usando protocolo bloqueo en 2 fases).

{ Duradera: si se termina con éxito (commit), los cambios en la BD son estables aunque luego falle otra

- Debe garantizarlo el método de recuperación.

# Recuperación



- | Caídas del sistema durante una transacción
- | Errores de ejecución: overflow, división por 0...
- | Errores lógicos que violan alguna regla de integridad (definida explícitamente o no) y que dejan inconsistente la BD -> programador/ABD
- | Problemas de concurrencia de transacciones
- | Fallos de lectura/escritura en disco
- | Problemas físicos y catástrofes: fuego, robos, sabotajes, fallos "humanos",... --> medidas de seguridad informática en la empresa.

# Recuperación



| Para que el sistema se pueda recuperar ante fallos se necesita grabar cada operación con la BD en un fichero LOG (bitácora). *Checkpoints*.

{ se escribe en el fichero LOG antes que en la BD

{ el fichero LOG debe estar en memoria estable

| Por cada operación se escribe un reg. en LOG

{ <**comienza-transacción**, numt>

{ <**escritura**, numt, id\_dato, val\_viejo, val\_nuevo>

{ <**lectura**, numt, id\_dato, valor>

{ <**termina\_transacción\_con\_éxito**, numt>

{ <**punto\_comprobación**, numt, numc>

# Problemas de concurrencia



- } La ejecución concurrente de transacciones puede dar lugar a problemas:
  - | Problema de la actualización perdida
  - | Problema de leer una actualización temporal (*lectura sucia*)
  - | Problema del resumen incorrecto
  - | Problema de la lectura no repetible

# Problemas de Concurrency



- } Sol. trivial: cada transacción se ejecuta en exclusión mutua. ¿Cuál sería la granularidad? ¿BD? ¿Tabla? ¿Tupla? ¿Atributo?
- } La solución trivial no es válida: muy restrictiva
  - | Se supone que las BDs se pensaron para que varios usuarios/aplicaciones accedieran a la vez
- } Hay que intercalar acciones pero que el resultado sea como en exclusión mutua

# Control de concurrencia: planes serializables

| Dadas las transacciones  $T_1, T_2, \dots, T_n$ ,

- $T_1$  compuesto por operaciones  $O_{11}, O_{12}, \dots, O_{1m_1}$
- $T_2$  compuesto por operaciones  $O_{21}, O_{22}, \dots, O_{2m_2}$
- ...  $T_n$  compuesto por operaciones  $O_{n1}, O_{n2}, \dots, O_{nm_n}$

| Un plan de ejecución concurrente de las transacciones sería:

- Ej:  $O_{11}, \underline{O_{21}}, O_{n1}, O_{n2}, O_{12}, \underline{O_{22}}, \dots, O_{1m_1}, \underline{O_{2m_2}}, \dots, O_{nm_n}$
- Una intercalación de todas las operaciones  $O_{ij}$  donde para todo  $i$ ,  $O_{i1}$  se ejecuta antes que  $O_{i2}$  ... antes que  $O_{imi}$

| Un plan es serializable si su resultado es el mismo que el producido por alguno de los posibles planes seriales de  $T_1, T_2, \dots, T_n$

- Ej:opers. de  $T_2$ , opers.  $T_1$ , opers.  $T_n$ , ..., opers. de  $T_3$

# Serializabilidad

- } Aparte de ACID, queremos que las transacciones sean serializables.
- } Determinar si un determinado plan es serializable es un problema NP-completo.
- } Solución: Imponer restricciones a la libre intercalación de operaciones entre transacciones
  - { Técnicas pesimistas: se impide realizar ciertas operaciones si son sospechosas de producir planes no serializables: BLOQUEOS (*lock*) y MARCAS DE TIEMPO (*time-stamping*)
  - { Técnicas optimistas: no imponen restricciones pero después se comprueba si ha habido interferencias

# Técnicas de bloqueo (lock)



- | A cada elemento de datos o gránulo X de la BD se le asocia una variable
  - { operación lock\_exclusivo(X): deja bloqueado al que lo pide si otro ya tiene cualquier lock sobre X
  - { operación lock\_compartido(X): deja bloqueado al que lo pide si otro ya tiene un lock exclusivo sobre X
  - { operación unlock(X): libera su lock sobre X
- | Antes de leer X → lock\_compartido(X)
- | Antes de escribir (leer) X → lock\_exclusivo(X)
- | Si no se va a leer o escribir más → unlock(X)

# Protocolo de Bloqueo en dos fases

- } Una transacción sigue el protocolo de bloqueo en dos fases si nunca hace un lock después de haber hecho algún unlock.
  - { Fase de crecimiento: se solicitan locks
  - { Fase de devolución: se realizan unlocks
- } Solamente este protocolo de bloqueo garantiza la serializabilidad de transacciones
- } Sin embargo, existe riesgo de deadlock !!
  - { Prevención de deadlocks
  - { Detección y recuperación de deadlocks

# Deadlocks



- } **Deadlock (o abrazo mortal o interbloqueo):** cuando una transacción T1 está bloqueada esperando a que otra T2 libere un lock, la cual también está bloqueada esperando a que T1 libere uno de sus lock. Se puede generalizar para N transacciones.
- } **Prevención de deadlocks**
  - | Cada transacción obtiene todos los locks al principio y si no puede entonces no obtiene ninguno. Problema de **livelock** (inanición de algunas transacciones que pueden no obtener todos los que necesitan)
  - | Los elementos de la BD están ordenados de alguna manera y los lock hay que obtenerlos en dicho orden. Los programadores deben controlarlo !!
- } **Detección y recuperación de deadlocks.**
  - | A medida que se piden y conceden los lock se construye un grafo de las transacciones que están esperando a otras. Si existe un ciclo en dicho grafo: deadlock. Hay que proceder a abortar a alguna de las transacciones. Problema de livelock si se aborta siempre a la misma!

# Técnicas de marcas de tiempo (*time-stamping*)

**Un timestamp es un identificador asignado a cada transacción TS(T). Indica la hora de comienzo de la transacción T. A cada elemento X de la BD se le asigna el timestamp de la última transacción que lo ha leído (TS\_lect(X)) y escrito (TS\_escr(X))**

**Si una transacción T quiere escribir en X**

- si  $TS\_lect(X) > TS(T)$  entonces abortar
- si  $TS\_escr(X) > TS(T)$  entonces no escribir y seguir
- en otro caso escribir y  $TS\_escr(X) := TS(T)$

**Una transacción T quiere leer de X**

- si  $TS\_escr(X) > TS(T)$  entonces abortar
- si  $TS\_escr(X) \leq TS(T)$  entonces leer de X y  $TS\_lect(X) := \text{máximo}(TS(T), TS\_lect(X))$

**Garantiza serializabilidad y ausencia de deadlocks.**

Puede haber livelock (si se aborta siempre a la misma transacción)

# Técnicas optimistas



- } No se realizan comprobaciones ANTES de ejecutar las operaciones (pedir locks, comprobar timestamps), sino al acabar toda la transacción (fase validación)
- } Durante la ejecución de la transacción se trabaja con copias
- } Hay tres fases en un protocolo optimista:
  - | Fase de lectura
  - | Fase de validación
  - | Fase de escritura
- } Es bueno cuando no hay muchas interferencias entre transacciones (por eso son "optimistas")

# Recuperación en Oracle



- } Redo logs (cíclicos)
- } Archive logs (consolidación de redo logs)
- } Backups
  - | Mirrors
  - | Export: Incremental, acumulativo (colección de incrementales), total
- } Recuperación basada en cambios, tiempo, paso-a-paso (basado en archive logs), completa

# Control de concurrencia en ORACLE (1)

- } Lectura consistente: garantiza que se lean los datos tal y como estaban al inicio de la transacción, sin impedir que otras transacciones los cambien.
  - | Implícitamente con `SELECT .. FROM T,R ...` (lo garantiza sobre las tuplas de T,R,...)
  - | Explícitamente con `SET TRANSACTION READ ONLY;` (lo garantiza sobre las tuplas de todas las tablas hasta el fin de transacción.)
    - { Debe ser la primera instrucción de la transacción
    - { No permitirá hacer ningún `INSERT`, `DELETE` o `UPDATE` en dicha transacción

# Control de concurrencia en ORACLE (2)

## } LOCKs

- | Explícitamente con `LOCK TABLE T IN x MODE`

- { x indica si sobre todas/algunas tuplas de T en modo compartido/exclusivo)

- | Implícitamente con cada operación (según cláusula `WHERE`)

- { `UPDATE, DELETE, INSERT`. Se bloquean las tuplas insertadas, borradas o actualizadas (al ser una transacción no finalizada)

- { `SELECT...FROM T FOR UPDATE OF atr.` Se bloquean las tuplas seleccionadas

# Control de concurrencia en ORACLE (y 3)

- } No hay UNLOCK explícitos en ORACLE!!
  - | Se realiza un UNLOCK implícito de todos los LOCK con cada COMMIT o ROLLBACK (implícitos o explícitos)
- } Pregunta: ¿Cómo conseguir que las transacciones en ORACLE sigan el protocolo en dos fases, o lo que es lo mismo, sean serializables?