Bases De Datos
Depto. Computación – FCEyN – UBA

Control de Concurrencia
Métodos Optimistas

Métodos Optimistas

- Timestamping
  - Timestamping multiversion
- Validación

Estos métodos asumen que no ocurrirá un comportamiento no serializable y actúan para reparar el problema sólo cuando ocurre una violación aparente.

Bibliografía: ***Database Systems. The Complete Book***. Second Edition. **Hector García-Molina, J.D. Ullman y Jennifer Widom**

Timestamping

- Cada transacción T tiene un único número llamado *timestamp*: Ts(T).Es asignado en orden ascendente.
  - Usar el reloj del sistema
  - El *scheduler* o planificador mantiene un contador.
    - Una transacción nueva que comienza siempre tiene un número mayor que una que comenzó antes.
  - El planificador debe mantener una tabla de las transacciones y sus *timestamps*.

Timestamping

- Cada elemento de la base de datos, X, debe asociarse a dos timestamp y un bit extra.
  - RT(X): tiempo de lectura, el timestamp más alto de una transacción que ha leído X
  - WT(X): tiempo de escritura, el timestamp mas alto de una transacción ha escrito X
  - C(X): bit de *commit* para X, es verdadero si y sólo si la transacción más reciente que escribió X ha realizado *commit.*

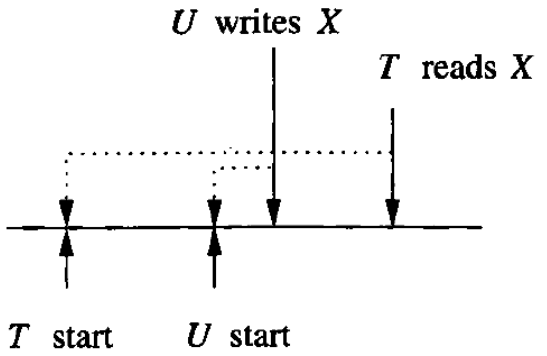
Comportamientos Físicamente Irrealizables

- El *planificador* asume que el orden de llegada de las transacciones es el orden serial en que deberían parecer que se ejecutan.

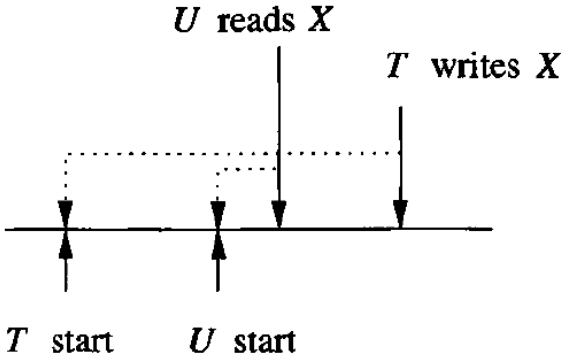
- El *planificador* además de asignar *timestamps* y actualizar RT, WT y C para cada elemento de una transacción debe verificar que cuando ocurre una lectura o escritura también podría haber ocurrido si cada transacción se hubiera realizado instantáneamente al momento del timestamp.
  - Si eso no ocurre entonces el comportamiento se denomina: **físicamente irrealizable**.

Comportamientos Físicamente Irrealizables

- *Read too late*.
  - Ts(T) < WT (X)
  - El valor de escritura indica que X fue escrito después de que teóricamente debería haberlo leído T.

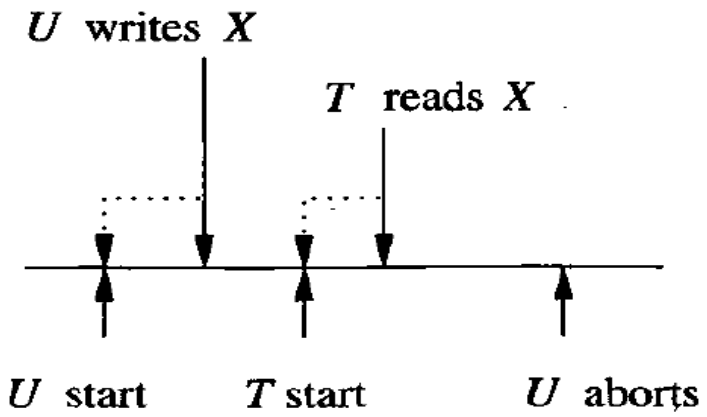


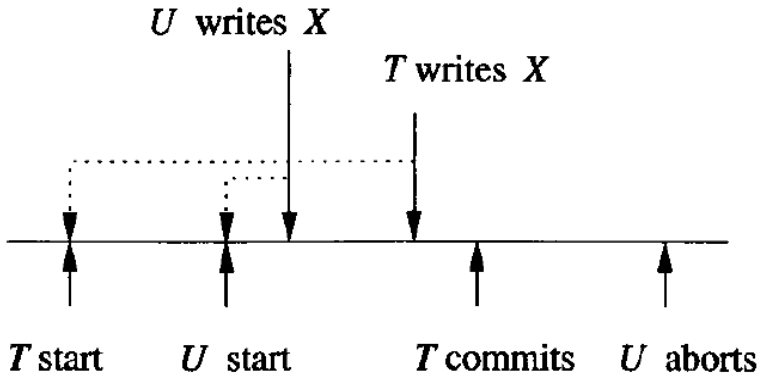
- *Write too late.*
  - WT (X) < Ts(T) < RT (X).
  - El tiempo de lectura indica que alguna otra transacción debería haber leído el valor escrito por T.



Dirty Data

- ***Dirty data*** usamos el bit de *commit*.
  - No es físicamente irrealizable.
  - Demoramos T hasta el commit o abort de U.
- *Thomas write rule*
  - *Problema si U aborta.*
- Cuando una transacción T escribe un elemento X, la escritura es tentativa y puede ser deshecha si T aborta. C(X) se pone falso y el planificador hace una copia de los valores de X y de WT (X) previos.





Reglas

- Ante la solicitud de una transacción T para una lectura o escritura, el planificador puede:
  - Conceder la solicitud
  - Abortar y reiniciar T con un nuevo *timestamp* (rollback)
  - Demorar T y decidir luego si abortar o conceder la solicitud (si el requerimiento es una lectura que podría ser sucia).

Reglas

1. El planificador recibe una solicitud $r_T(X)$
   a) Si *Ts (T) >= WT(X)* - *es físicamente realizable*
      i. Si **C(X) es True**, conceder la solicitud. Si TS(T) > RT(X) hacer RT (X)=Ts (T) sino no cambiar RT (X).
      ii. Si **C(X) es falso** demorar T hasta que C(X) sea verdadero o la transacción que escribió a X aborte.
   b) Si *Ts (T) < WT (X)* - *es físicamente irrealizable* **Rollback T** (abortar y reiniciar con un nuevo timestamp)

Reglas

2. El planificador recibe una solicitud $w_T(X)$
   a) Si *Ts (T) >= RT(X) y Ts (T) >= WT (X) - es físicamente realizable*
      i. Escribir el nuevo valor para X
      ii. $W_T(X)$:= Ts (T) Asignar nuevo W_T a X.
      iii. C(X):= false. Poner en falso el bit de commit.
   b) Si *Ts (T) >= $R_T(X)$ pero Ts (T) < $W_T(X)$ es físicamente realizable* pero ya hay un valor posterior en X.
      i. Si **C(X) es true**, ignora la escritura.
      ii. Si **C(X) es falso** demorar T hasta que C(X) sea verdadero o la transacción que escribió a X aborte
   c) Si Ts(T) < RT (X) entonces *es físicamente irrealizable*

Reglas

3. Si el planificador recibe una solicitud de Commit(T).
   a) Para cada uno de los elementos X escritos por T se hace
      - C(X) := true.
      - Se permite proseguir a las transacciones que esperan a que X sea committed.

4. Si el planificador recibe una solicitud de Abort(T) o Rollback(T)
   a) Cada transacción que estaba esperando por un elemento X que T escribió debe repetir el intento de lectura o escritura y verificar si ahora el intento es legal

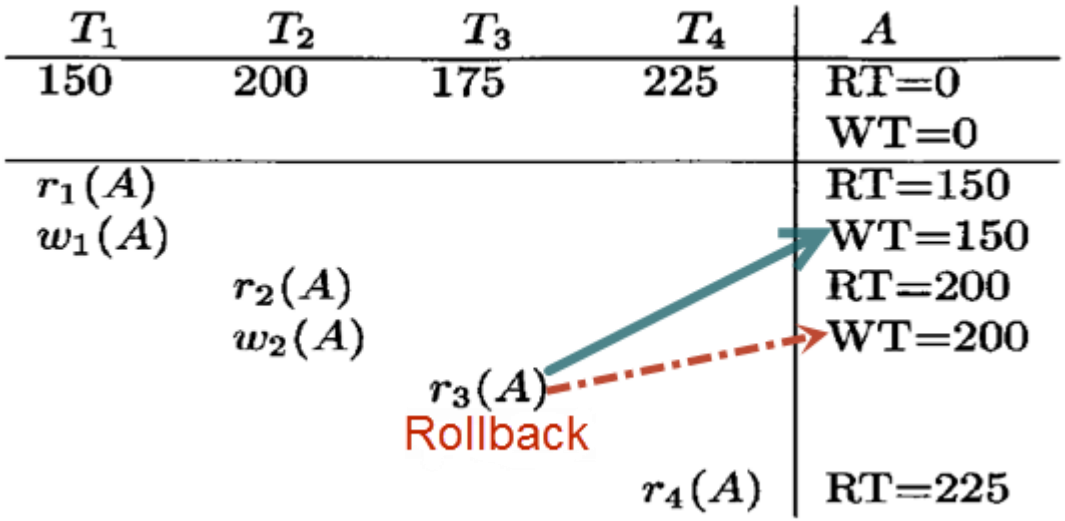
Ejemplo

T_1	T_2	T_3	A	B	C
200	150	175	RT=0 WT=0	RT=0 WT=0	RT=0 WT=0
$r_1(B)$;				RT=200	
	$r_2(A)$;		RT=150		
		$r_3(C)$;			RT=175
$w_1(B)$;				WT=200	
$w_1(A)$;			WT=200		
	$w_2(C)$;				
	Rollback				
		$w_3(A)$;			

Timestamps Multiversión

- Mantiene versiones antiguas de los elementos de la base de datos
- Permite *RT(X) que en otras ocasiones causarían que la transacción T aborte debido a que la versión actual de X fue escrita por una transacción posterior.*
- *Permite leer la versión de X apropiada según el timestamp de T.*

Timestamps Multiversión



- T3 debería realizar *rollback* al no poder un leer valor de X que fue escrito por una transacción posterior

Planificador multiversión

- Cuando ocurre $w_T(X)$, ***si es legal*** *entonces se crea* ***una nueva versión*** *del elemento X. Su tiempo de escritura es Ts(T) y nos referimos a él como Xt, donde t = Ts(T).*
- *Cuando ocurre una lectura $r_T(X)$ el scheduler busca una versión Xt de X tal que t <= Ts(T) y que no haya otra versión Xt' tal que t < t' <= Ts(T).*
- *Los tiempos de escritura están asociados a versiones de un elemento y nunca cambian.*
- *Los tiempos de lectura también son asociados con versiones. Lo podemos notar como Xt,tr (donde tr es el último tiempo de lectura de Xt) . Una transacción T' debe hacer rollback cuando existe alguna Xt,tr tal que t < Ts(T') y tr > Ts(T')*
  - *Ejemplo: si tenemos X50 y X100. Una transacción T con Ts(T)=80 lee X50. Una transacción T' con Ts(T') =60 intenta escribir. En este caso T' debe hacer rollback.*
- *Si una versión Xt tiene un tiempo de escritura tal que no existe una transacción activa T tal que Ts(T) sea menor t, se pueden borrar cualquier versión de X previa a Xt.*

Multiversión vs. No Multiversión

T_1	T_2	T_3	T_4	A
150	200	175	225	RT=0
				WT=0
$r_1(A)$				RT=150
$w_1(A)$				WT=150
	$r_2(A)$			RT=200
	$w_2(A)$			WT=200
		$r_3(A)$		
		Rollback		
			$r_4(A)$	RT=225

T_1	T_2	T_3	T_4	A_0	A_{150}	A_{200}
150	200	175	225			
$r_1(A)$				Read		
$w_1(A)$					Create	
	$r_2(A)$				Read	
	$w_2(A)$					Create
		$r_3(A)$			Read	
			$r_4(A)$			Read

Compromiso

- Transacciones *read-only* vs. Transacciones *read-write*.
- Transacciones *read-write* se manjean con locking pero crean versiones de los elementos.
- Transacciones *read-only* se manejan con versiones creadas por transacciones *read-write*

Validación

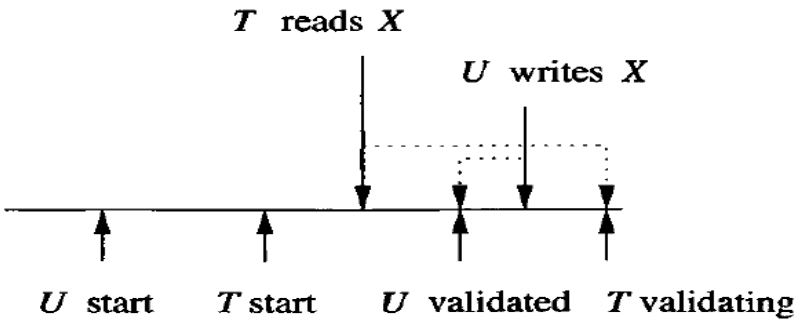
- Se debe tener para cada transacción T los conjuntos:
  - RS(T) elementos leídos por T.
  - WS(T) elementos escritos por T.
- Transacciones se ejecutan en 3 fases:
  1. **Lectura**: Lee desde la base de datos todos los elementos en su RS(T). Calcula en su espacio de direcciones local los elementos a escribir.
  2. **Validación**: el planificador valida la transacción comparando su RS y WS con los de otras transacciones. Si la validación falla se ejecuta un rollback y se reinicia, sino se pasa al paso 3.
  3. **Escritura**: Los elementos de WS son escritos en la base de datos.

Validación

- El *planificador* mantiene tres conjuntos:
  - START: conjunto de transacciones que han comenzado pero aún no completaron la validación. Para cada transacción T en este conjunto se mantiene START(T) que es el tiempo en el cual T comenzó.
  - VAL: el conjunto de transacciones que han sido validadas pero aún no finalizaron la fase de escritura. Para cada transacción T en este conjunto se mantienen dos valores START(T) y VAL(T). Este último es el tiempo en el cual T es validada
  - END: el conjunto de transacciones que han completado la fase 3. El *planificador* mantiene para estas transacciones START(T), VAL(T) y END(T). Este conjunto que crecería indefinidamente puede ser limpiado eliminado aquellas transacciones T tales que para cualquier transacción activa U pase que END(T) < START(U)
- El orden serial puede pensarse usando el tiempo de validación. Es decir la transacción T debería ejecutarse en el momento de su validación formando un orden serial hipotético.

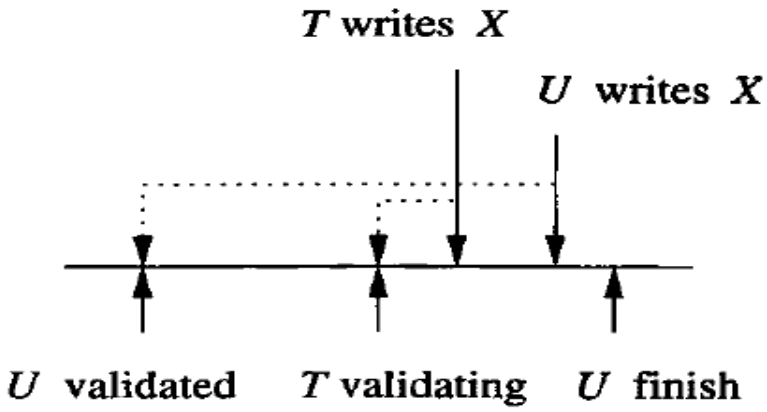
¿Qué puede ir mal?

- Supongamos una transacción U y una transacción T tal que.
  - *U esta en VAL o END; o sea: U fue validada*
  - *END(U) > START(T), U no terminó antes que el comienzo de T*
  - RS(T) ∩ WS(U) no es vacío.



¿Qué puede ir mal?

- Supongamos una transacción U y una transacción T tal que.
  - *U esta en VAL. **U** fue validada exitosamente.*
  - *END(U) > VAL(T). **U** no finalizó antes de que **T** haya entrado en su fase de validación.*
  - WS(T) ∩ WS(U) no es vacío. Por ejemplo X está en ambos conjuntos de escritura.



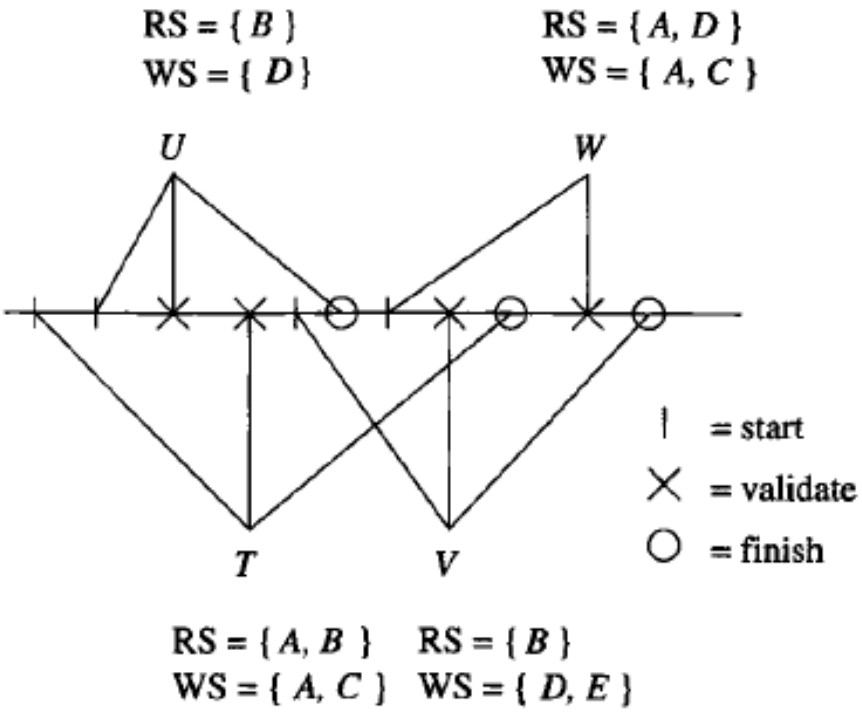
T **no puede validarse** exitosamente **si podría llegar a escribir algo antes** que una transacción anterior

Regla para Validación

- Para validar una transacción T hay que:
  - Verificar que RS(T) ∩ WS(U) es vacío para cualquier transacción U validada previamente y que no finalizo antes de que T ***comience***.
  - Verificar WS(T) ∩ WS(U) es vacío para cualquier transacción U validada previamente y que no finalizo antes de que T ***sea validada***.

Ejemplo

- 4 transacciones T, U, W y V



❖Verificar que RS(T) ∩ WS(U) es vacío para cualquier transacción U validada previamente y que no finalizo antes de que T comenzara.

❖Verificar WS(T) ∩ WS(U) es vacío para cualquier transacción U validada previamente y que no finalizo antes de que T sea validada.

Transacciones en SQL

Transacciones Implícitas/Explicitas

- ISO SQL: cualquier comando SQL al comienzo de una sesión o inmediato posterior al fin de una transacción comienza automáticamente una nueva transacción (DB2 y Oracle)
- SQL Server, MySQL/InnoDB, PostgreSQL funcionan por defecto en modo AUTOCOMMIT
  - MySQL/InnoDB: SET AUTOCOMMIT = {0|1}
  - SQL Server: SET IMPLICIT_TRANSACTIONS [ON| OFF}

Transacciones Implícitas/Explicitas

- BEGIN/START TRANSACTION;
- COMMIT;
- ROLLBACK;

```
INSERT INTO Tabla (id, s) VALUES (1, 'primero');
INSERT INTO Tabla (id, s) VALUES (2, 'segundo');
INSERT INTO Tabla (id, s) VALUES (3, 'tercero');
SELECT * FROM Tabla ;

ROLLBACK;
SELECT * FROM Tabla ;
```

Cuidado!!!

- Supongamos lo siguiente:

```
create table Cuentas(ctaID INTEGER NOT NULL PRIMARY KEY, balance
decimal(11,2) CHECK (balance >= 0) );

START TRANSACTION;
UPDATE Cuentas SET balance = balance - 100 WHERE ctaID = 101;
UPDATE Cuentas SET balance = balance + 100 WHERE ctaID = 102;
COMMIT;
```

SQLSTATE

- (ISO-89)SQLCODE → (ISO-92)SQLSTATE
- 5 Caracteres :
  - Clase 2 Caracteres
  - Subclase 3 Caracteres
- '00000' ÉXITO
- '40…' Transacción perdida.

Niveles de aislamiento

El nivel de aislamiento controla el grado en que una transacción dada está expuesta a la acciones de otras transacciones ejecutándose simultáneamente.

Fenómenos

- Lost Update
- Dirty Read
- Non-Repeatable Read (Fuzzy Read)
- Phantom Read

Niveles de Aislamiento

- SET TRANSACTION ISOLATION LEVEL READ UNCOMMITED;

Isolation level	Dirty read	Nonrepeatable read	Phantom
Read uncommitted	Yes	Yes	Yes
Read committed	No	Yes	Yes
Repeatable read	No	No	Yes
Serializable/ *Snapshot*	No	No	No

Niveles de Aislamiento

- PostegreSQL:
  - SET TRANSACTION modo
  - SET SESSION CHARACTERISTICS AS TRANSACTION modo
  - MODO:
    - ISOLATION LEVEL { SERIALIZABLE | REPEATABLE READ | READ COMMITTED | READ UNCOMMITTED } READ WRITE | READ ONLY
  - Default: READ COMMITTED (READ UNCOMMITTED es tratado como READ COMMITTED)

Snapshot vs Serializable

- SQL SERVER

set transaction isolation level SERIALIZABLE/SNAPSHOT
begin tran
update marbles set color = 'White' where color = 'Black'
---
commit tran
select * from marbles

set transaction isolation level SERIALIZABLE/SNAPSHOT
begin tran
update marbles set color = 'Black' where color = 'White'
commit tran

¿Preguntas?

- ¿Preguntas?