# Control de Concurrencia

BD Consistente=Cuando todas las restricciones (constraints) para todos los datos de la BD son satisfechas[1].

T=Conjunto de operaciones que se hacen con los datos de una BD[1].

# Ejemplo de T

Alicia tiene dos cuentas bancarias, X e Y, en X tiene $ 1500 y en Y tiene $ 500. Quiere transferir $ 500 de X a Y.
Cuando la T se complete, ambas cuentas deberán tener $ 1000, cualquier otro valor implica una BD INCONSISTENTE.

# Propiedades de una T

Para asegurar una BD CONSISTENTE, las T's deben satisfacer 4 propiedades:

**A**tomicidad=se hacen todas las Operaciones de la T o ninguna de ellas.

**C**onsistencia=la T debe ser escrita correctamente por el programador.

**I**solation (aislación)=la T debe correr sin interferencias de otras T's.

**D**urabilidad=los cambios que hizo la T deben persistir, incluso ante fallos (se persiste todo o nada).

# El Ejemplo de Alicia

X=item de dato que representa el saldo de la cuenta X

Y=item de dato que representa el saldo de la cuenta Y

# El Ejemplo de Alicia

La T podria escribirse como:

```
Begin-Tran T1:
        Read(X)
        Read(Y)
        X=X-500
        Y=Y+500
        Write(X)
        Write(Y)
End-Tran T1;
```

# El Ejemplo de Alicia

Resumiendo, La T podria escribirse como:

Begin-Tran T1:
R1(X)
R1(Y)
W1(X)
W1(Y)
End-Tran T1;

O bien, también:

T1=R1(X),R1(Y),W1(X),W1(Y)

# El Ejemplo de Alicia

Agreguemos ahora la T2 que pretende sumar $ 200 al saldo de la cuenta Y de Alicia, la T2 podria escribirse como:

Begin-Tran T2:
R2(Y)
Y=Y+200
W2(Y)
End-Tran T1;

O bien, y resumiendo, también:

T2=R2(Y),W2(Y)

# Ejemplo Alicia: ejecutando T1, T2 sin aislación (Isolation)

Recordemos (X=$1500,Y=$500)
- T1 transfiere $500 de X a Y
- T2 suma $200 a Y

Ejecución posible 1:
R1(X),R1(Y),R2(Y),W1(X),W1(Y),W2(Y)
Resultado Inconsistente: X=$1000, Y=$700

Ejecución posible 2:
R1(X),R1(Y),R2(Y),W1(X),W2(Y),W1(Y)
Resultado Inconsistente: X=$1000, Y=$1000

# Sistema Control Concurrencia

La inconsistencia que pueden generar T1, T2 ocurre por no haber aislación (Isolation) entre T1 y T2.

La aislación debe implementarse a través de un Sistema de Control de Concurrencia (SCC).

SCC se implementa utilizando Lockeos (L) o Timestamping (ts).

# Ciclo de Vida de una T

## Definición de T de Jim Gray.

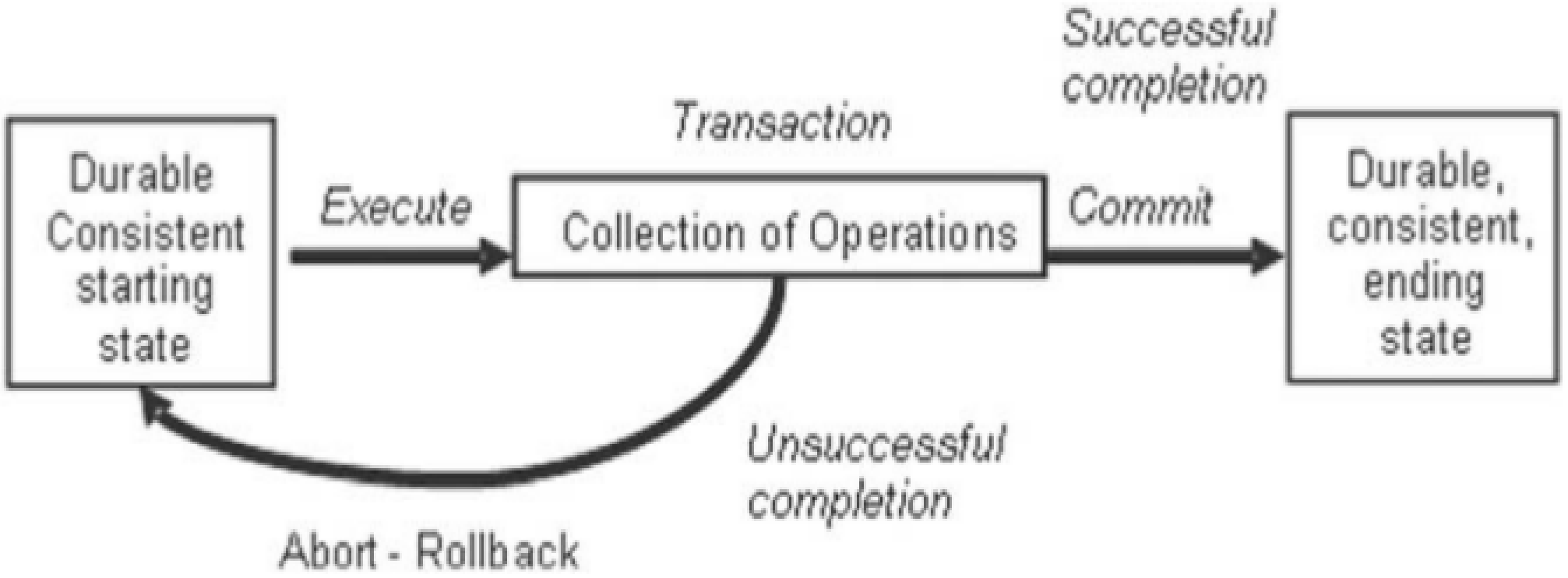



**Figure 5.2** Transaction life cycle.

# Planificación (Schedule) de T's

**Planificador Serial (PS)**=las Op de las T's no se solapan en el tiempo(t) → en un t dado solo 1 T esta en ejecución → (-) performance, no aceptable hoy día

**Planificador Paralelo (PP)**=las Op de las T's se solapan en el tiempo (t) → en un t dado N T's en ejecución → Op conflictivas sobre un mismo item de dato, (+) performance

# Operaciones (Op) Conflictivas sobre un mismo item de dato

| | Read T2 | Write T2 |
|---|---|---|
| Read T1 | No Conflicto | Conflicto |
| Write T1 | Conflicto | Conflicto |

# Planificacion Equivalente de T's

Op conflictivas -> anomalías

Anomalías → inconsistencia BD

Planificador → preservar consistencia BD

Planificador → si serie no aceptable → paralelo → producir planificaciones equivalentes (PE)

PE=Dos planificaciones son PE si ambas producen el mismo estado de la BD, leen los mismos valores y escriben los mismos valores.

# Ejemplo PE
# S1=P paralela, S2=P serie

S1=R1(X),W1(X),R2(X),W2(X),R1(Y),W1(Y),R2(Y),W2(Y)

S2=R1(X),W1(X),R1(Y),W1(Y),R2(X),W2(X),R2(Y),W2(Y)

T1 lee X y graba X, antes de que lo lea T2, idem Y → producir los mismos valores.

**S1 es serializable. S1, S2 producen una BD consistente. S1,S2 son equivalentes.**

# Orden de Commit

"El orden de 2 Op conflictivas de 2 T's distintas determina el orden de commit en una planificación serializable equivalente (PSE)"

Ej: R1(X),W2(X) → primero hacer commit de T1 y luego de T2

2 T's pueden tener Op conflictivas más de una vez, el SCC va armando un orden parcial de commit (PCO).

Una P es PS sí y solo sí, ninguno de los PCO's son contradictorios.

PCO's no contradictorios → graficado → orden total de commit (TCO) → es un gráfico acíclico.

# Orden de Commit, Ejemplos

S1=R1(X),R2(X),W2(X),W1(X)

| Conflicto | Orden de Commit |
|---|---|
| R1(X),W2(X) | T1->T2 |
| R2(X),W1(X) | T2->T1 |
| W2(X),W1(X) | T2->T1 |

Grafico cíclico, S1 no serializable

S2=R2(X),R1(X),W2(X),W3(X)

T2->T3, T1->T2, T1->T3, T2->T3

T1->T2->T3 grafico acíclico, S2 serializable y equivalente a:

R1(X),R2(X),W2(X),W3(X)

# Seriabilidad CDBE

"Una planificación es serializable sí y sólo sí es equivalente a una planificación serial"

# Seriabilidad DDBE

"Si hay -al menos- una planificación local (PL) no serializable, entonces la planificación global (PG) es no serializable"

# Seriabilidad DDBE

Si todas las PL son serializables, entonces

Si BD es no replicada, entonces

PG serializable

sino

Requerimiento de consistencia mutua (todas las copias del item deben tener el mismo valor)-> PG es serializable sí y sólo sí todas las PL son serializables y el orden de commit de 2 T's en conflicto es el mismo en cada sitio en donde se ejecutan

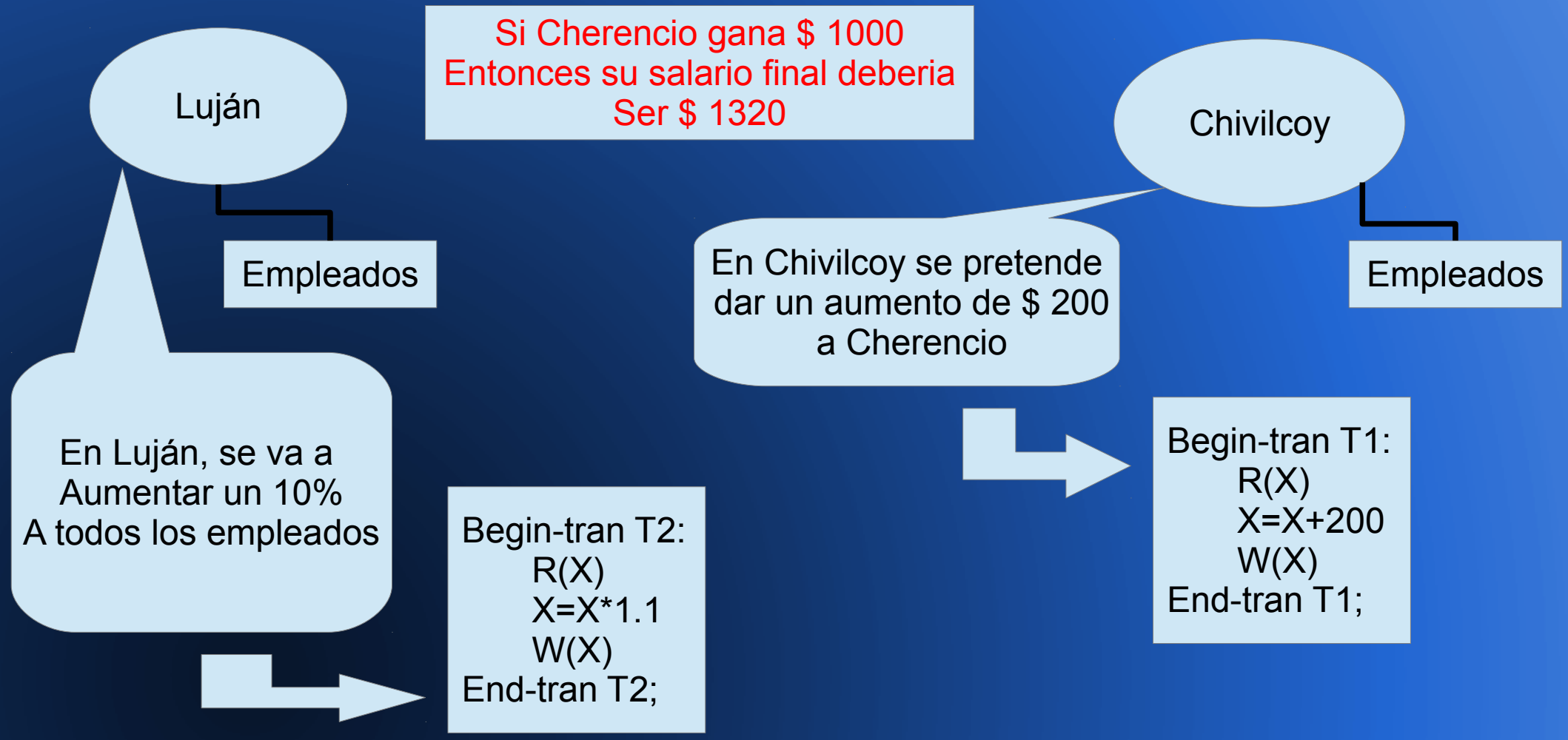
Fin-si

Sino

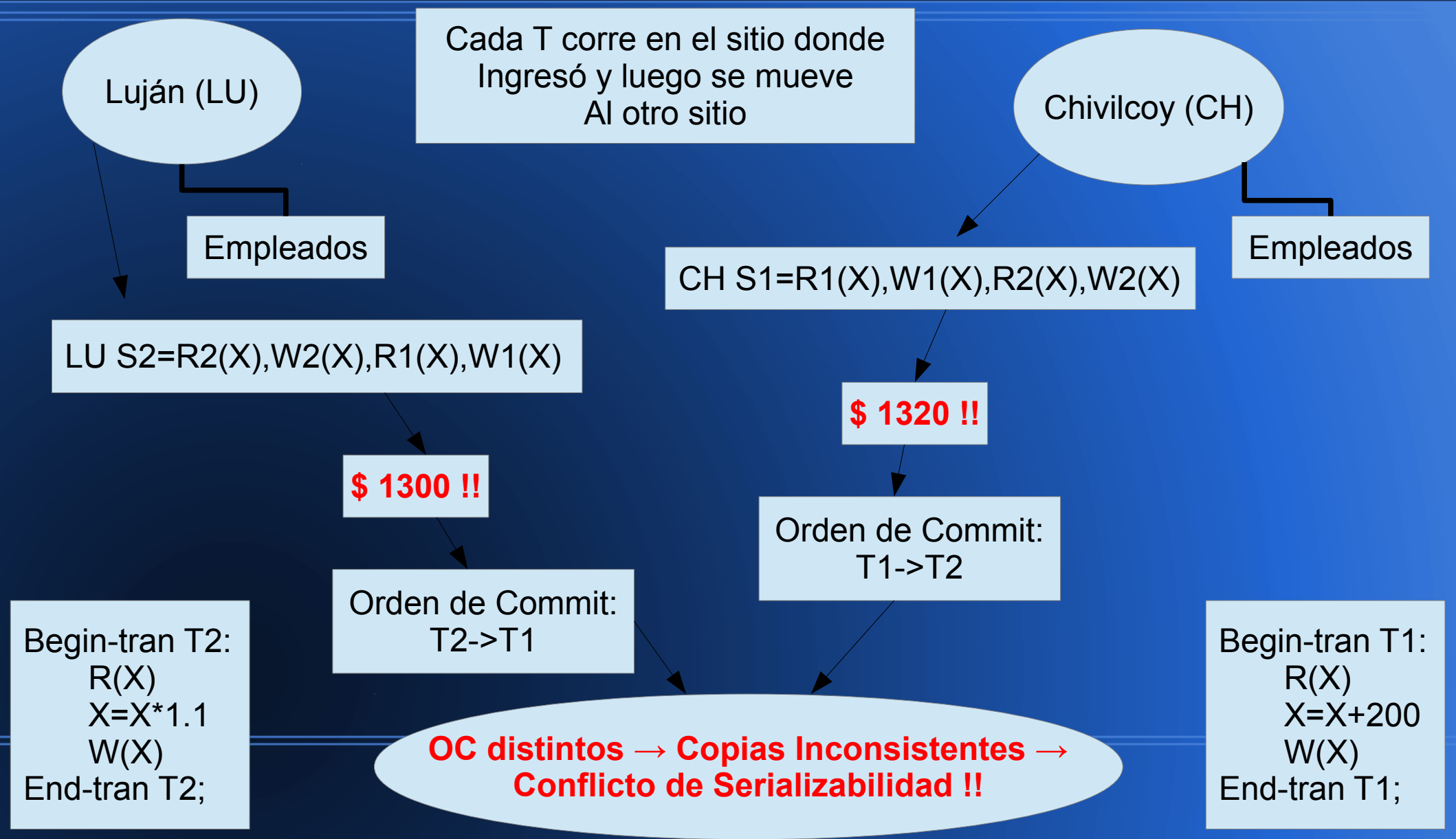
PG no serializable

Fin-si

# Seriabilidad DDBE, Ejemplo



# Seriabilidad DDBE, Ejemplo



# Planificaciones Recuperables

Permiten recuperar la BD a un estado consistente luego del fallo de una o más T's

Ej: **S=R1(X),W1(X),R2(X),W2(X),C2,R1(Y)**

S es conflicto serialización en T1->T2 , en caso de hacer commit de T1 luego de T2, pero si T1 aborta, lo escrito por T2 es inválido y se debe abortar también a T2 y toda otra T que haya leído lo escrito por T2, también deben ser abortadas! (rollback en cascada).

Recuperar la BD al estado anterior de S es difícil o casi imposible! Porque S es una planificación no recuperable.

Recuperar una BD a un punto en el tiempo se llama **PIT** (point in time) o **PITR** (point in time recovery).

Para evitar PIT, se puede rechazar S o bien aceptar S pero demorar el commit de T2 hasta que lo haga T1:

**S=R1(X),W1(X),R2(X),W2(X),R1(Y),C1,C2**

# Control Centralizado de Concurrencia

**2 Enfoques básicos para la Aislación de T's en CBE**

- Locking
- Timestamping

**Algoritmos de Control De Concurrencia Segun Serialización**

- Pesimistas
- Optimistas

# Algoritmos de Control de Concurrencia

| Pesimistas | Optimistas |
| --- | --- |
| Alta tasa de conflictos | Baja tasa de conflictos |
| Identificar conflictos y sincronizar T's | Tratan de demorar la sincronización de T's |
| Re-arranque frecuente de T's | Re-arranque infrecuente de T's |

# Algoritmos basados en lock's

| **Matriz de compatibilidad de Lock's**<br>Determina si un item puede ser lockeado por mas de una T al mismo tiempo | | |
|---|---|---|
| | **Read Lock** | **Write Lock** |
| **Read Lock** | **Permitido** | **No Permitido** |
| **Write Lock** | **No Permitido** | **No Permitido** |

# Alg. Lockeo 1 pasada (1PL)

Cada T pone lock sobre item a usar y lo libera tan pronto como ha finalizado de usarlo.

**Problema: NO PRODUCE PLANIFICACIONES SERIALIZABLES**

Por que?: porque no mantiene el lock el t suficiente para asegurar serialización

**RLj(X)=Read lock sobre X en Tj**

**WLj(X)=Write lock sobre X en Tj**

**LRj(X)=Release lock sobre X en Tj**

# Ejemplo 1PL

S=WL1(X),R1(X),W1(X),LR1(X),RL2(X),R2(X),LR2(X),WL2(Y),
R2(Y),W2(Y),LR2(Y),WL1(Y),R1(Y),W1(Y),LR1(Y)

==>

S=R1(X),W1(X),R2(X),R2(Y),W2(Y),R1(Y),W1(Y)

==>

T1->T2, T2->T1

==>

S NO SERIALIZABLE

# Alg. Lockeo 2 pasadas (2PL)

Cada T no entrelaza su adquisición de lockeo y release entre distintas T's.

**1era Fase: la T solo adquiere lock y no libera ninguno, hasta que todos los locks hayan sido otorgados ==> fase de crecimiento**

**2da Fase: la T comienza liberando lockeos y no requiere más lockeos ==> fase de encogimiento**

# Ejemplo 2PL

**S=WL1(X),R1(X),W1(X),WL1(Y),LR1(X),R1(Y),W1(Y),LR1(Y),RL2(X),R2(X),WL2(Y),LR2(X),R2(Y),W2(Y),LR2(Y)**

T1 no libera el lock de X hasta después del lock de Y

**==>**

**S=R1(X),W1(X),R1(Y),W1(Y),R2(X),R2(Y),W2(Y)**

**==>**

**T1->T2, T1->T2**

**==>**

**S ES SERIALIZABLE**

# Alg. Basados en TimeStamp

Timestamp=antiguedad de la T=ts(T)=fecha nacimiento de la T,clock CPU,contador unico incremental (nro de T)

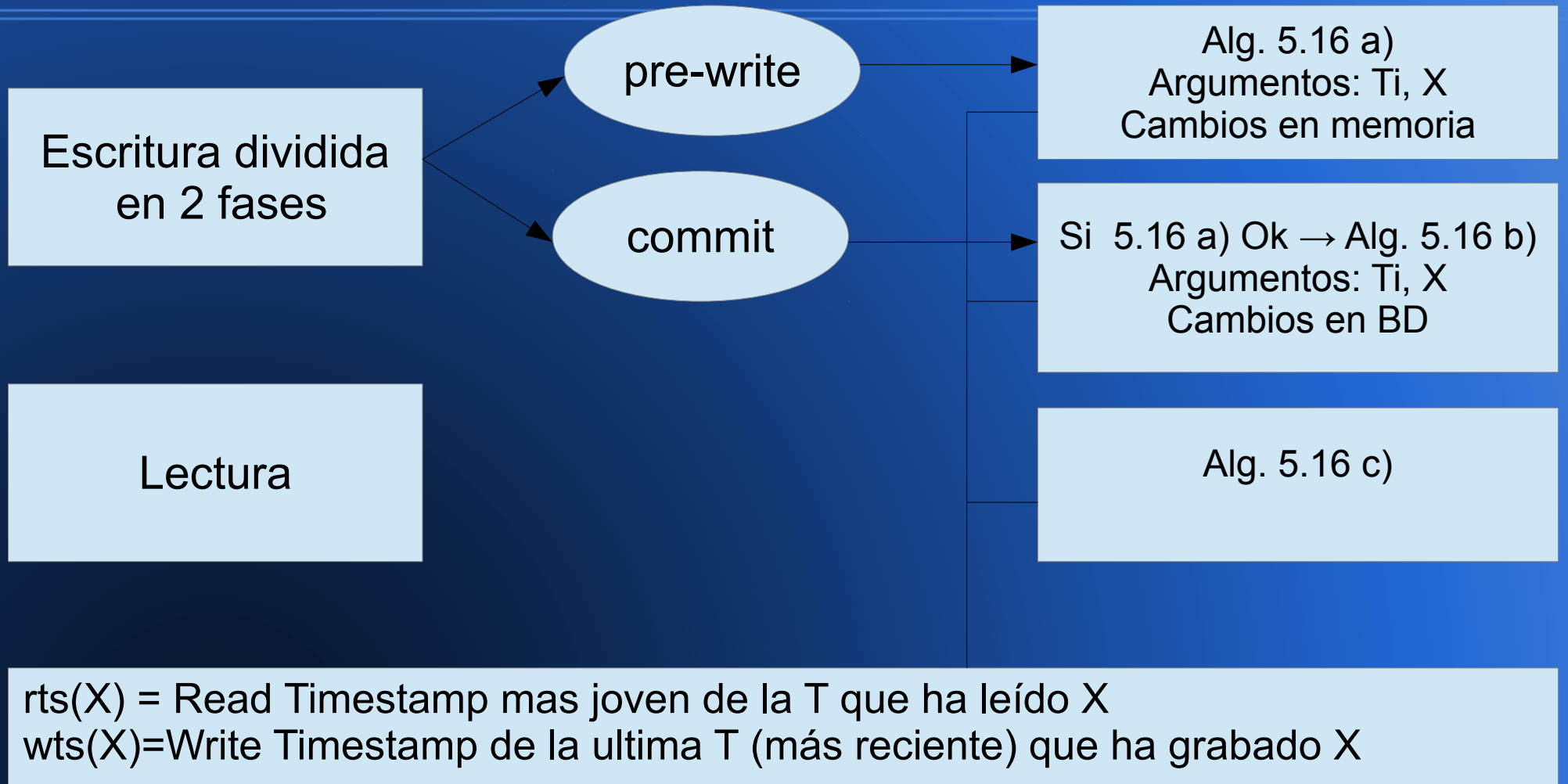
3 reglas para reforzar seriabilidad:

**Regla de Acceso=cuando 2 T's acceden al mismo t el mismo item, siempre ingresa la T mas vieja, forzando a la T mas nueva a esperar hasta que la T mas vieja haga commit**

**Regla T tardía=una vieja T no tiene permitido leer o grabar un item que esta siendo grabado por una T mas joven**

**Regla de espera de T joven=una T joven puede leer o grabar un item que ha sido escrito por una T vieja.**

# Alg. Basados en TimeStamp



# Alg. Basados en TimeStamp

**5.16 a)**

```
Begin pre-write (Ti,X)
    If ts(Ti) < rts(X) or ts(Ti) < wts(X) then
        /* X ha sido accedido por una T mas joven */
        /* Ti es muy tardia */
        Rollback Ti;
        Restart Ti  (con un nuevo ts mas tarde)
    Else
        Cambia buffer para Ti con ts(Ti)
        /* ahora Ti esta pendiente de commit */
    End-If
End pre-write
```

# Alg. Basados en TimeStamp

**5.16 b)**

```
Begin commit (Ti,X)
    /* Ti intenta hacer commit de X */
    Para todo Tj más viejo y pendiente
        Ti espera hasta que Tj haga commit o aborte
        Sino
            Ti commit X
            wts(X) = ts(Ti)
    End-Para
End commit
```

# Alg. Basados en TimeStamp

**5.16 c)**

```
Begin read (Ti,X)   /* Ti intenta leer X */
    If ts(Ti) < wts(X) then
        /* X ha sido escrito por una T mas joven */
        Rollback Ti
        Reiniciar Ti mas tarde con un ts mas nuevo
    Sino
        Para todo Tj mas vieja y pendiente
            Ti espera hasta que Tj haga commit o aborte
        Sino
            Ti lee X
            rts(X) = max(ts(Ti),rts(X))
        End-Para
    End-If
End read
```

# Alg. Multi-versión (MV)

**En vez de grabar un único valor de X, el sistema mantiene múltiples X, cada T que graba X, genera una nueva versión de X → el Sistema no tiene que bloquear la lectura de X → Ti encuentra un X escrito por Tj, tal que, Tj < ts(Ti) , siendo Tj la T más joven que grabó X.**

Se rechaza Ti para grabar X si hay otra Tj mas joven que Ti que ha leído X y pretende grabar X.

Cada X tiene su rts, wts

# Alg. Multi-versión (MV)

```
Begin Multi-Version(Ti,X)
    /* xk es 1 versión X tal que wts(xk) es la
    Última versión para ts <= ts(Ti)  */
    If Operación Ti es read then
        Ti lee xk
        rts(xk)=max(ts(Ti),rts(xk))
    Sino if Operación Ti es write then
        If ts(Ti) < rts(xk) then
            Rollback Ti
            Re-start Ti con nuevo ts
        Sino
            Crear nueva versión de X (xm)
            rts(xm)=ts(Ti)
            wts(xm)=ts(Ti)
        End-if
    End-if
End Multi-Version
```

# Alg. Optimista

**Tasa de conflictos baja → demorar control de seriabilidad justo antes del commit (+performance)**

Ciclo de Vida T en 3 fases:

Fase Ejecución (EP)=T hace sus Op's en memoria

Fase Validación (VP)=T se valida a sí misma para asegurarse que su commit no rompe la integridad de la BD

Fase Commit (CP)=T graba sus cambios de memoria a disco.

rs(Tj)=conjunto de items que Tj lee

ws(Tj)=conjunto de items que Tj graba

# Alg. Optimista

**3 reglas para asegurar serialización durante etapa VP:**

Regla 1: Para Ti, Tj; donde Ti lee lo que graba Tj, la EP de Ti no puede solaparse con la CP de Tj. Tj arrancará su CP después de que Ti finalice su EP.

Regla 2: Para Ti, Tj; donde Ti graba lo que Tj lee, la CP de Ti no puede solaparse con la EP de Tj. Tj arrancará su EP después de que Ti finalice su CP.

Regla 3: Para Ti,Tj; donde Ti graba lo que Tj graba, la CP de Ti no puede solaparse con la CP de Tj. Tj arrancará su CP después que Ti finalice su CP.

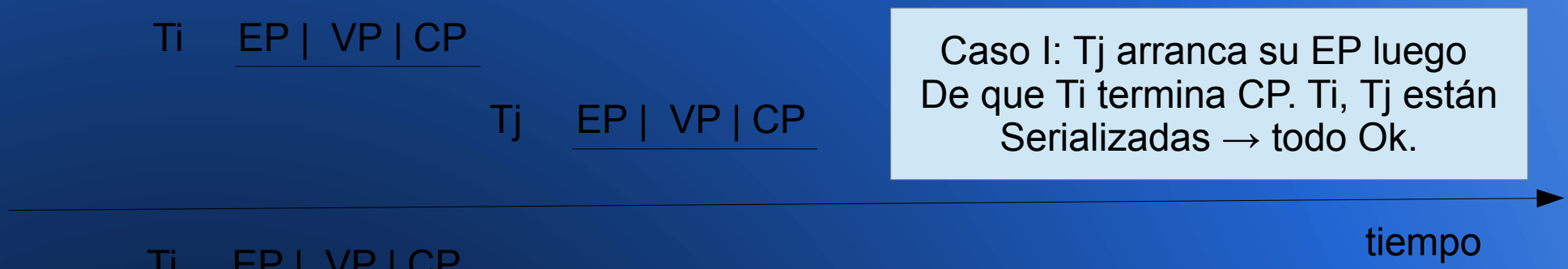
# Alg. Optimista

El ts de una T se asigna recién cuando la T esta lista para VP.

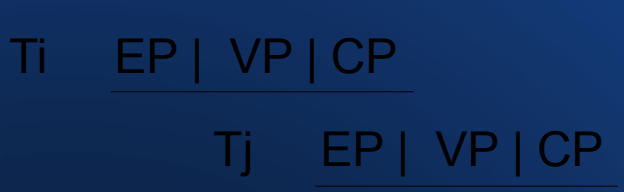
Si Tj completa su EP, Tj puede ser comiteada luego de haber sido validada contra todas las otras T que estan corriendo.

# Alg. Optimista

Hay 3 casos que cubren todos los posibles conflictos:




Caso I: Tj arranca su EP luego De que Ti termina CP. Ti, Tj están Serializadas → todo Ok.




Caso II: Tj arranca CP después De CP de Ti → asegurarse que: ws(Ti) ∩ rs(Tj) es conjunto vacío


Caso III: Tj termina su EP Después que Ti termina su EP → asegurarse que ws(Ti) ∩ (rs(Tj) u ws(Tj)) es conjunto vacío

# Control de Concurrencia en BDD (SCCD)

SCC en CDBE esta en 1 sitio.
SCC en DDBE esta en N sitios.
SCCD puede ser centralizado o distribuido.
SCCD=serialización local+serialización global.
TD=T distribuida

2 Requerimientos para Conj TD's:
1) Todas las PL deben ser serializables
2) Si 2 TD's estan en conflicto en mas de un sitio, sus PCO's en todos los sitios deben ser compatibles con sus conflictos.

# Alg. 2PL en BDE

En BDE no hay un administrador de lockeos (LM) centralizado. Una T entra en Sitio 1 y pide lockeo al Sitio 5.

3 variantes de 2PL:

- **2PL Centralizado**
- **2PL Copia Primaria**
- **2PL Distribuido**

# 2PL Centralizado

**Un sitio se designa como administrador central de lockeos (CLM). Cada sitio obtiene locks de CLM, CLM envía mensaje de "lock granted" cuando el ítem esta disponible o bien encola la T para esperar por el ítem. Si el TM local recibió "lock granted", entonces puede continuar. Cuando la T termina, el TM envía "lock release" al CLM, éste libera ítem y otorga lock a la primera T en espera por el ítem.**

**(-) sobrecarga sitio con CLM, si cae CLM cae todo el sistema.**

# 2PL Copia Primaria

**N sitios designados como CLM's. Cada CLM administra lockeos sobre distintas tablas. Cada sitio sabe dónde se encuentra cada tabla-> cuando un TM necesita pedir lock dirige la petición al CLM apropiado.**
**Si hay N copias de 1 tabla, el TM decide que copia leer y se asegura que todas las copias sean escritas en caso de una operación de escritura.**

# 2PL Distribuido

Cada administrador de lockeo (LM) reside en el lugar en donde se encuentra el item a administrar. Hay 3 alternativas:

-BDE no tiene datos replicados: LM reside en dónde se encuentra la única copia del item

-BDE con todos los datos replicados: uno de los sitios es elegido como LM para cada item. Si el LM se centraliza en 1 solo sitio->2PL Centralizado

-BDE parcialmente replicado: LM reside en sitio en donde residen items no replicados. Para items replicados se elije 1 sitio como LM para cada item. TM debe adquirir lock del LM apropiado. TM responsable de reflejar update en todos los sitios en donde el item reside.

# 2PL Distribuido

**Una vez que el LM es elegido, el enfoque es el mismo para las 3 alternativas. Permite planificaciones serializadas pero no evita el interbloqueo.**

# Alg. Optimista Distribuido

**Se aplican 2 reglas:**
**1) validar T localmente: Ti validada en todos los sitios donde corra (usando alg optimista local). Si Ti es inválida en 1 o más sitios->Ti abortada. Ti necesita validación global.**
**2) validar T globalmente: cuando 2 T's conflictivas corren en más de un sitio, se requiere que el orden de commit sea el mismo en todos los sitios->El commit de Ti es demorado hasta que todas las T's conflictivas previas a Ti en el orden de serialización han si comiteadas o abortadas->el alg se termina haciendo pesimista!**
**Solución: mantener un orden total de commit global, todos los planif locales envían a un TM global su orden de commit cuando validan localmente la T. El TM global solo valida la T si el orden de commit en todos los sitios es compatible, sino aborta la T.**

# Bibliografía


- [1] Saeed K. Rahimi, Frank S. Haug, "Distributed Database Managment Systems: A Practical Approach", Wiley-IEEE Computer Society Press., 2010, Chapter 5 Concurrency Control