Transacciones

Universidad de Oviedo

Este material está adaptado principalmente del curso *CS145 Introduction to Databases* impartido por **Christopher Re** en **Universidad de Stanford** (https://cs145-fa18.github.io/)

Escuela de Ingeniería Informática

2022-2023

Section 1

Introducción

Precuela

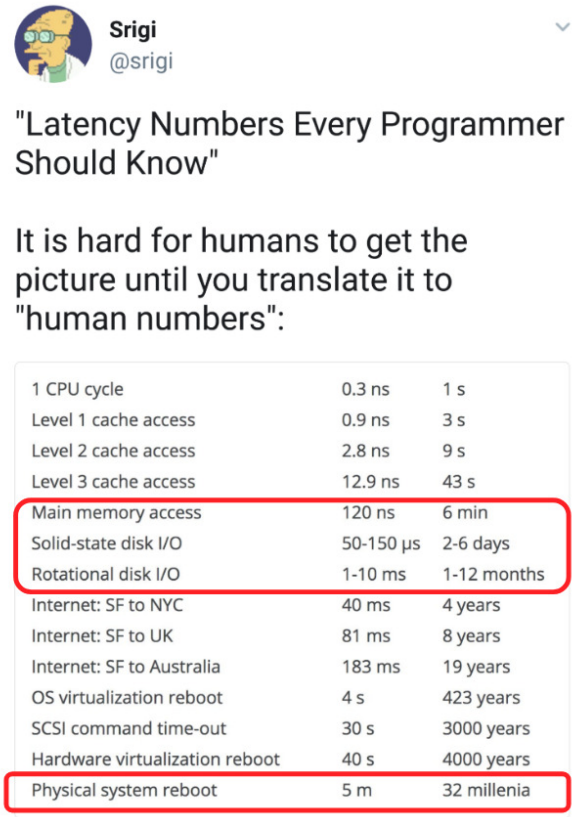
Almacenamiento y rendimiento

Blog de Peter Norvig norvig.com

Latency numbers every engineer should know

Ballpark timings on typical PC:

execute typical instruction	1/1,000,000,000 sec = 1 nanosec
fetch from L1 cache memory	0.5 nanosec
fetch from L2 cache memory	7 nanosec
Mutex lock/unlock	25 nanosec
fetch from main memory	100 nanosec
send 2K bytes over 1Gbps network	20,000 nanosec
read 1MB sequentially from memory	250,000 nanosec
fetch from new disk location (seek)	8,000,000 nanosec
read 1MB sequentially from disk	20,000,000 nanosec
send packet US to Europe and back	150 milliseconds = 150,000,000 nanosec



Srigi
@srigi

"Latency Numbers Every Programmer Should Know"

It is hard for humans to get the picture until you translate it to "human numbers":

1 CPU cycle	0.3 ns	1 s
Level 1 cache access	0.9 ns	3 s
Level 2 cache access	2.8 ns	9 s
Level 3 cache access	12.9 ns	43 s
Main memory access	120 ns	6 min
Solid-state disk I/O	50-150 µs	2-6 days
Rotational disk I/O	1-10 ms	1-12 months
Internet: SF to NYC	40 ms	4 years
Internet: SF to UK	81 ms	8 years
Internet: SF to Australia	183 ms	19 years
OS virtualization reboot	4 s	423 years
SCSI command time-out	30 s	3000 years
Hardware virtualization reboot	40 s	4000 years
Physical system reboot	5 m	32 millenia

- **Memoria**: Acceso **rápido** pero con **capacidad limitada** y **volátil**.
- **Disco**: Acceso *lento* pero con **gran capacidad** y **persistente**.

¿Cómo podemos utilizar ambas eficazmente y con garantías?

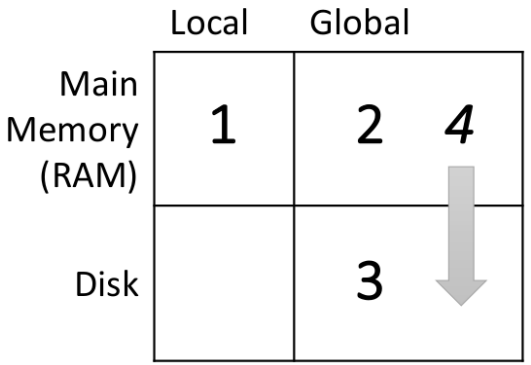
Ejecución concurrente para mejorar el rendimiento.

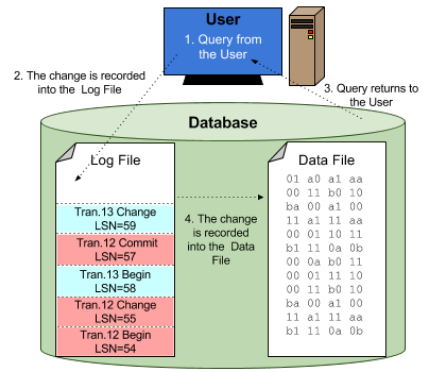
Motivación para los mecanismos que presentamos en este tema.

Organización de la memoria asociada con un SGBD

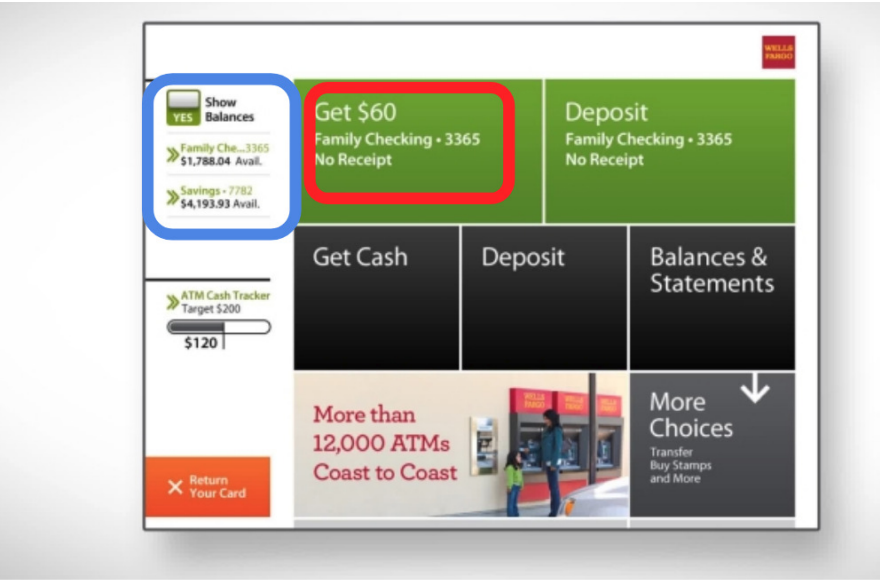
Tres tipos de regiones de memoria

1. **Local**: Cada proceso tiene su propia area de memoria local de acceso privado.
2. **Global** o **compartida**: Los procesos pueden leer de o escribir en memoria global compartida (**Database Buffer Cache**).
3. **Disco**: La memoria global puede ser leída desde o volcada (flush) a disco.
4. **Log**





Sacar dinero de un cajero automático



Read Balance
Give money
Update Balance

vs

Read Balance
Update Balance
Give money

Requisitos:

- Resistencia a fallos del sistema, como fallos de hardware o caídas del sistema.
- Durabilidad, por lo que si la operación tiene éxito, la nueva información perdurará en el tiempo.
- Acceso simultáneo a la base de datos para lograr un mejor rendimiento.

Otras lecturas [1]

Transacciones

Una transacción (*txn*) es una secuencia de una o más operaciones (lectura/escritura) que refleja una única operación en el mundo real.

Si el sistema falla, los cambios de cada transacción se reflejan en su totalidad o no se reflejan en absoluto.

```
START TRANSACTION
        UPDATE Product
        SET Price = Price – 1.99
        WHERE pname = 'Gizmo'
COMMIT
```

- Transferencia de dinero entre cuentas.
- Comprar un conjunto de productos.
- Inscribirse en una clase (o a una lista de espera).

- Por defecto, cada sentencia SQL individual se ejecuta de forma transaccional (aunque inserten, actualicen o eliminen miles de filas). Valor por defecto de la propiedad `AutoCommit = true`
- Se puede cambiar para agrupar varias sentencias en una misma transacción `AutoCommit = false`

```
UPDATE Bank SET amount = amount - 100 WHERE name = 'Bob'
```

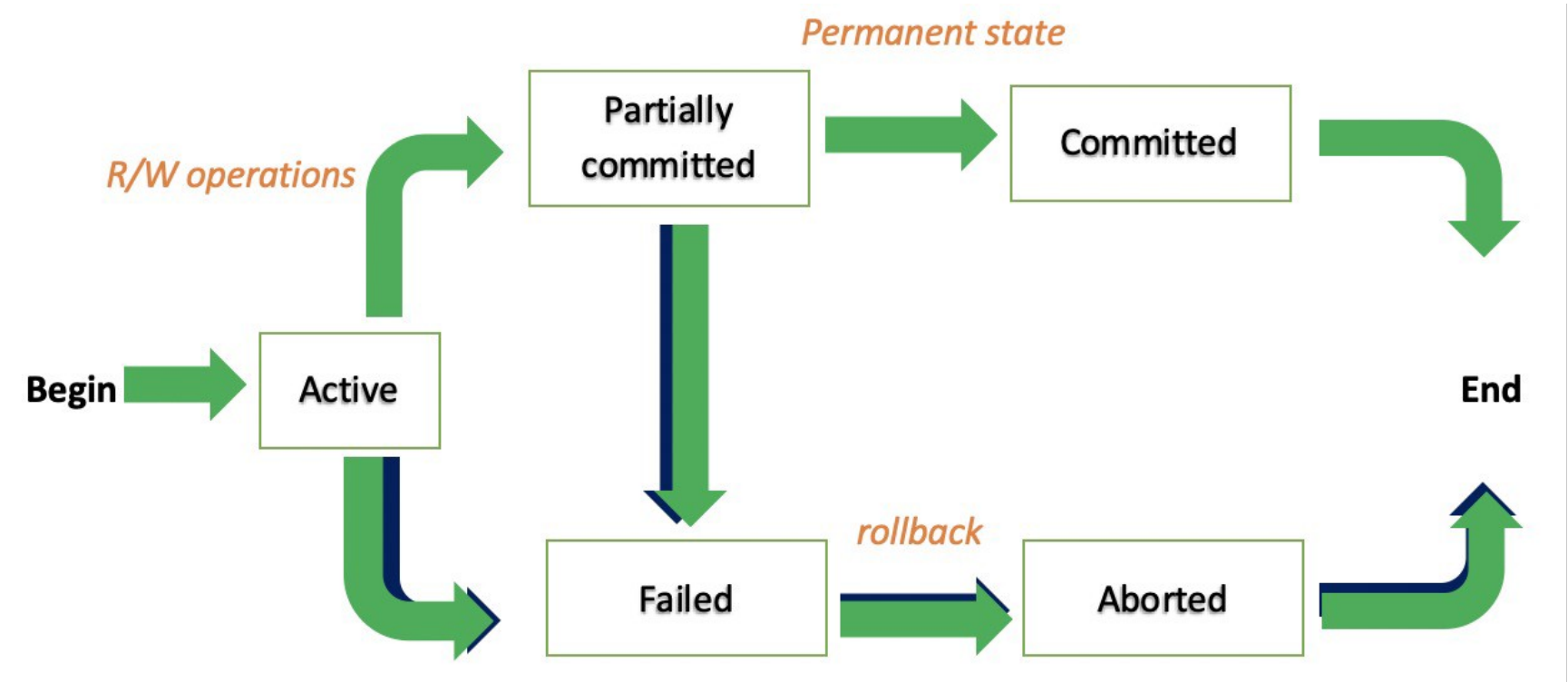
```
UPDATE Bank SET amount = amount + 100 WHERE name = 'Alice'
```

```
SET AUTOCOMMIT OFF
UPDATE Bank SET amount = amount - 100 WHERE name = 'Bob'
UPDATE Bank SET amount = amount + 100 WHERE name = 'Alice'
```

- Una transacción comienza implícitamente con cada conexión.
- Una transacción finaliza (y comienza una nueva) cuando:
  - termina la conexión actual.
  - **Commit** implícito o explícito. Finaliza la transacción actual (vuelca al disco) y comienza una nueva.
  - **Rollback** hace que la transacción actual se cancele.

```
SET AUTOCOMMIT OFF
UPDATE Bank SET amount = amount - 100 WHERE name = 'Bob'
UPDATE Bank SET amount = amount + 100 WHERE name = 'Alice'
COMMIT
```

```
SET AUTOCOMMIT OFF
UPDATE Bank SET amount = amount - 100 WHERE name = 'Bob'
UPDATE Bank SET amount = amount + 100 WHERE name = 'Alice'
ROLLBACK
```

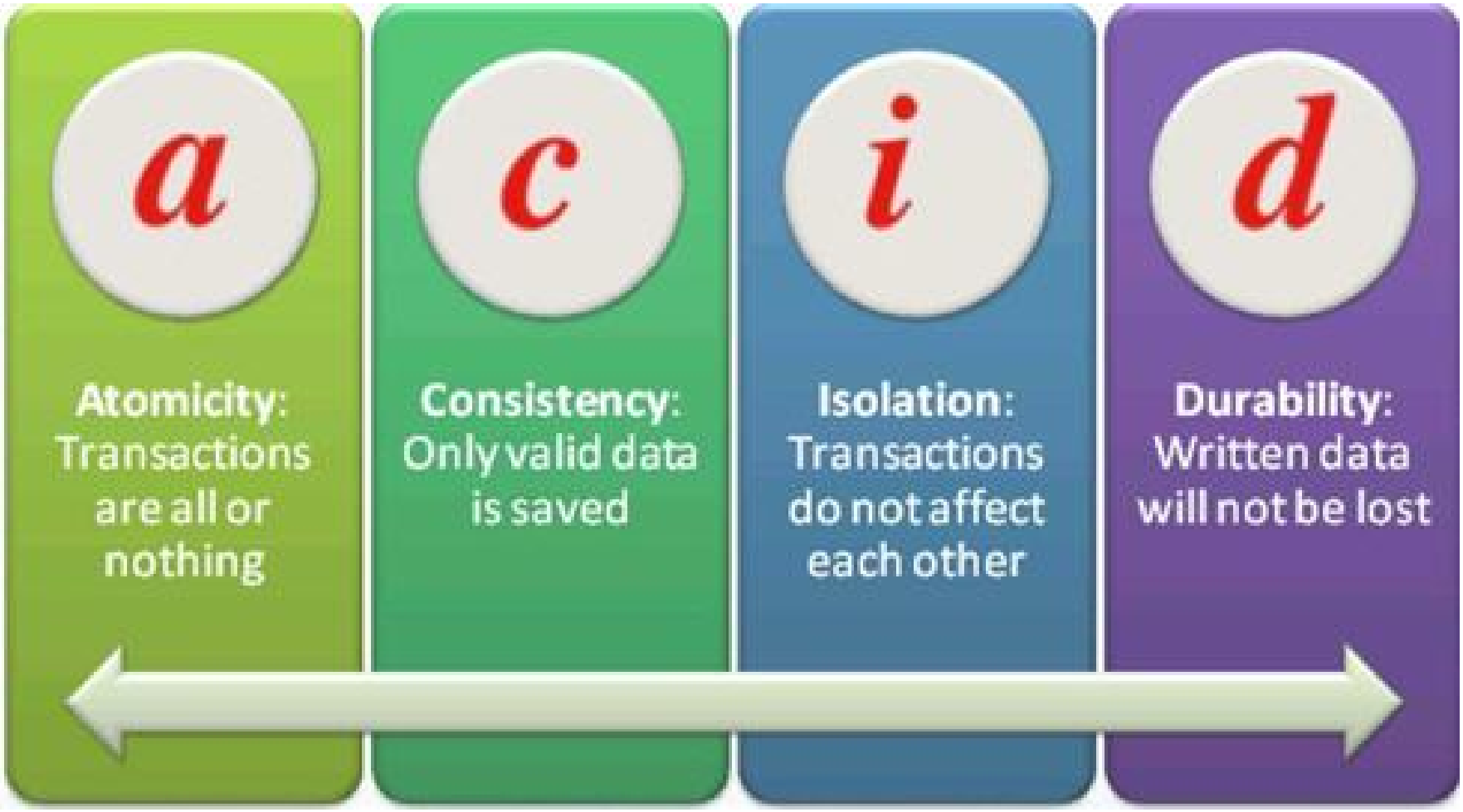

Permanent state
Partially committed
Committed
R/W operations
Begin
Active
End
rollback
Failed
Aborted

Section 2

Propiedades de las transacciones

ACID

Guía rápida del comportamiento esperado de las transacciones



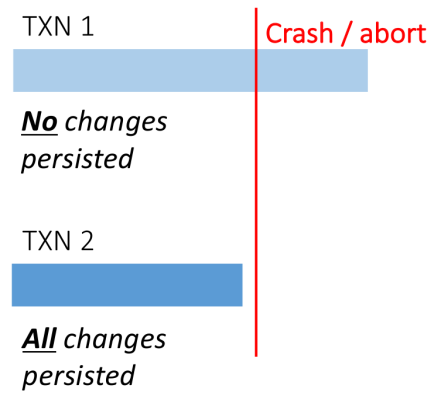
ACID: Atomicidad

Las instrucciones agrupadas en una txn son atómicas: **todas o ninguna**

- Intuitivamente, en el mundo real, una transacción es algo que ocurriría *completamente o no ocurriría*

Una transacción puede acabar de dos formas:

- **commit**, **confirmar**: Se hacen efectivos los cambios
- **abort**: se descartan los cambios; **rollback**



Ejemplo: Si durante una transacción bancaria se realiza la primera acción (débito) pero, por algún fallo del sistema, no la segunda (crédito), los valores no se reflejarían.

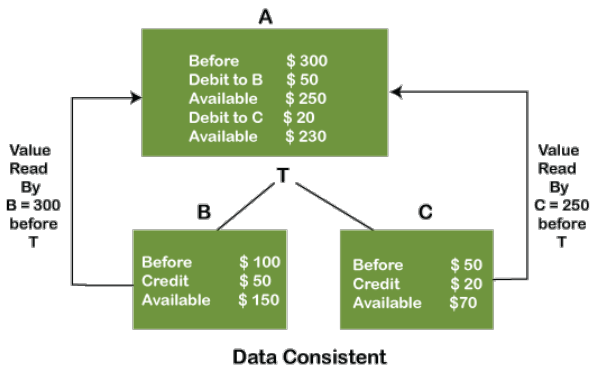
A**C**ID: **C**onsistencia

Cualquier transacción debe conducir la BBDD de un estado consistente a otro estado consistente.

- **Durante**: se admiten **temporalmente** datos inconsistentes
- Al **finalizar**: sólo datos **consistentes** que cumplan las restricciones.

Ejemplos restricciones:

- Las claves primarias y las claves candidatas son únicas; por ejemplo, el número de pasaporte (no PK)
- En una transferencia, la cantidad total de dinero no cambia.
- No se admiten valores negativos.

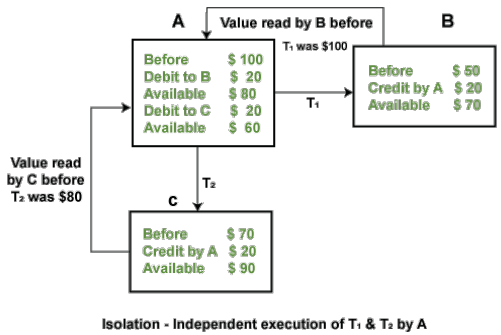


ACID: AIslamiento

Aunque varias txn se ejecuten concurrentemente, el efecto final debe ser el mismo que si **cada txn se ejecutase una tras otra**

- Una txn no debería poder observar cambios parciales de otras transacciones durante la ejecución.
- Ejemplo
  - Joe y Mary ejecutan txn contra una bbdd al mismo tiempo.
  - Ambas txn deben operar en la bbdd de manera aislada.

- Se debe ejecutar **completamente** la txn de Joe antes que la de Mary o viceversa.
- O SGBD maneja los detalles de *intercalar* varias TXN para que lo parezca.
  - Evita que la txn de Joe lea datos intermedios producidos como **parte** de la txn de Mary.
  - Puede **no comprometer** finalmente los datos.



Isolation - Independent execution of T₁ & T₂ by A

ACI**D**: **D**urability

- Una vez que una transacción ha sido **confirmada, committed**, sus efectos deben ser permanentes, incluso si después falla el sistema.
- Mientras una transacción está en curso (**parcialmente confirmada**), los efectos no son persistentes.
- Si algo falla, se restaurará la BBDD a un estado consistente antes de que comience la transacción.

Section 3

Concurrencia, Planificación y Serialización

Escribir transacciones

Una transacción es una lista de operaciones.
Las **operaciones** son

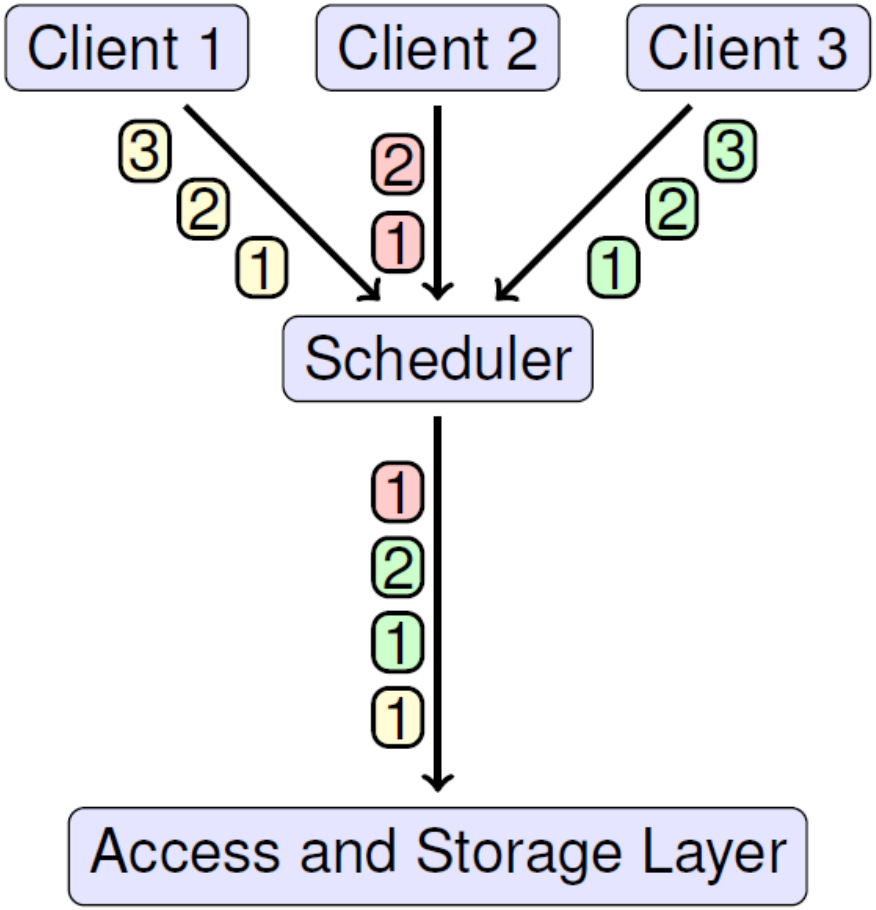
- lecturas (**R(O)**) y
- escrituras (**W(O)**)

de objetos O de la base de datos.
Las transacciones terminan con **Commit** o **Abort**[a].

[a] A veces se omiten si no son relevantes

Ejemplo:
T_1 : `R(V), R(Y), W(V), W(C), Commit`

Planificación o *Schedule*



- Una **planificación** es el intercalado de operaciones de un conjunto de transacciones.
- El **planificador** decide el **plan de ejecución, planificación**
- El orden en una planificación de dos operaciones de una txn T debe ser en el mismo orden en que aparecen en T.

$$T_1 : \text{R(V) W(V)}$$
$$T_2 : \text{R(Y) W(Y)}$$

Which of the following is a schedule of these transactions?

S_1 :

T_1	R(V)			W(V)
T_2		R(Y)	W(Y)	

S_2 :

T_1	W(V)			R(V)
T_2		R(Y)	W(Y)	

Planificación serie y planificación serializable

Una planificación es serie si las operaciones de las transacciones no se intercalan sino que se ejecutan una tras otra.

S_1 :

T_1			R(V)	W(V)
T_2	R(Y)	W(Y)		

Una planificación es serializable si su efecto en la BBDD es el mismo que alguna planificación serie.

Cuestión planificación serializable

Habitualmente, es deseable una planificación serializable, ¿ por qué?

Considere dos transacciones

Cada operación lee un valor (de la memoria global), realiza alguna operación con él (que puede cambiarlo) y luego vuelve a escribirlo.

```
T1: START TRANSACTION
        UPDATE Accounts
        SET Amt = Amt + 100
        WHERE Name = 'A'

        UPDATE Accounts
        SET Amt = Amt - 100
        WHERE Name = 'B'
COMMIT
```

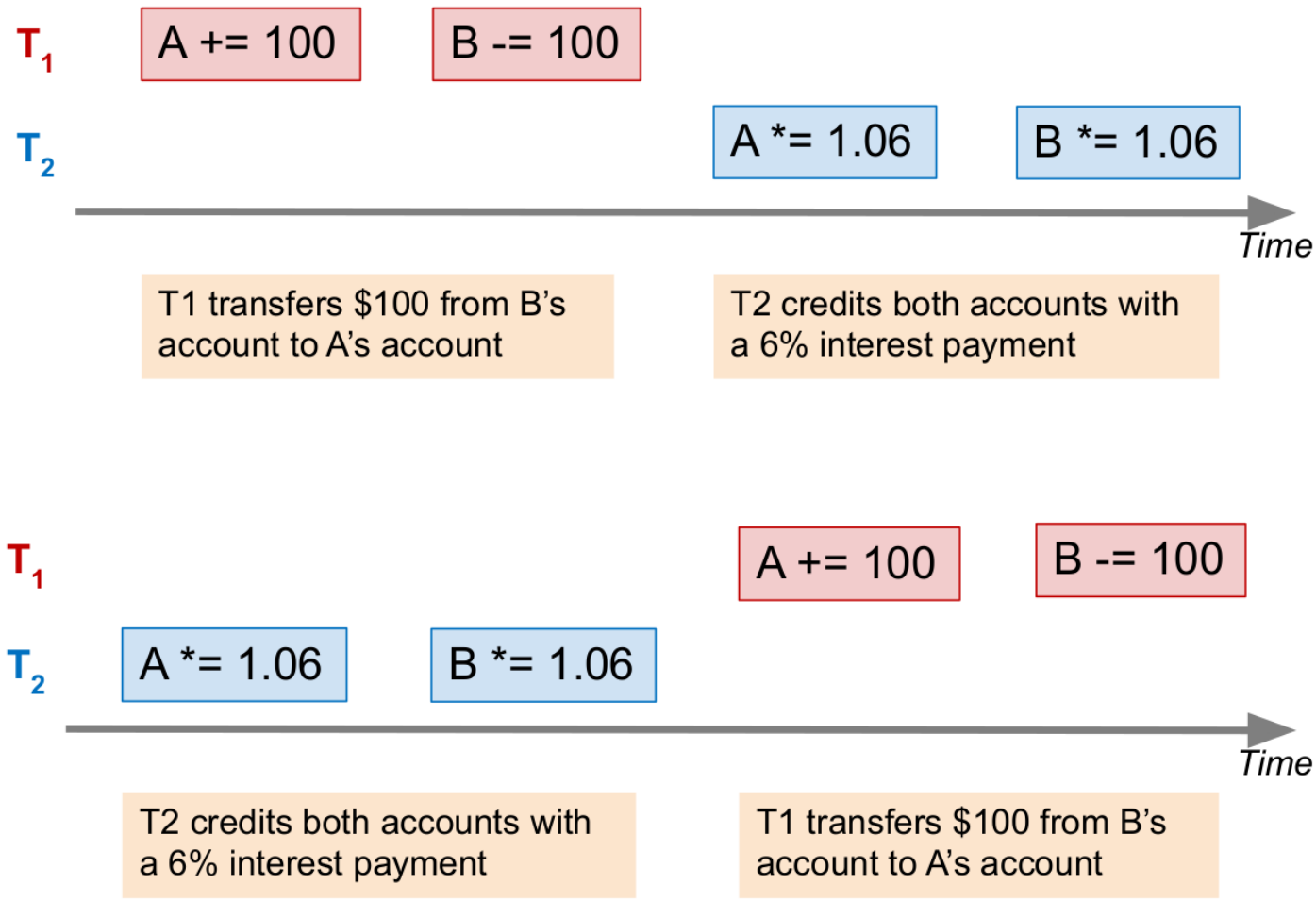
T1 transfers $100 from B's account to A's account

```
T2: START TRANSACTION
        UPDATE Accounts
        SET Amt = Amt * 1.06
COMMIT
```

T2 credits both accounts with a 6% interest payment

Veamos las TXN en una vista cronológica

Ejecución serie

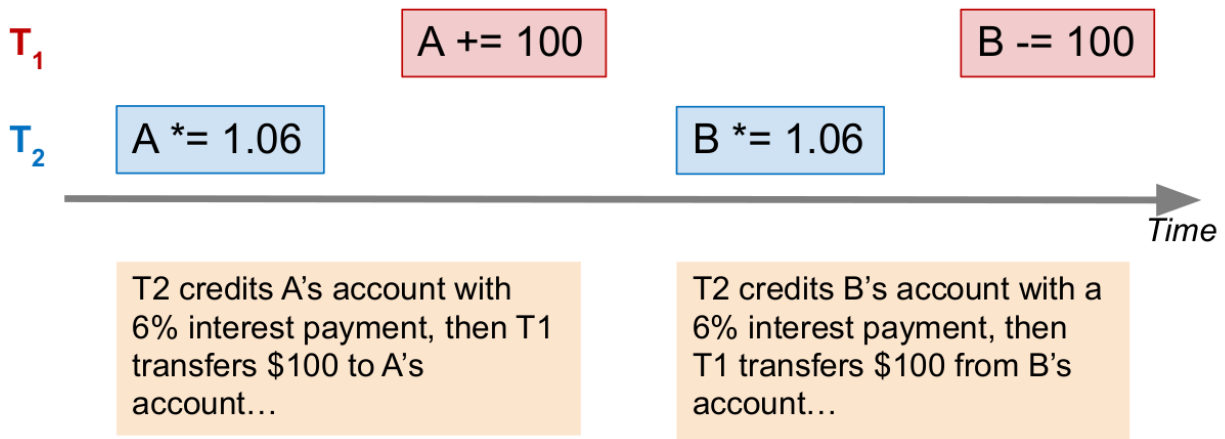


Intercalar operaciones para aumentar el rendimiento

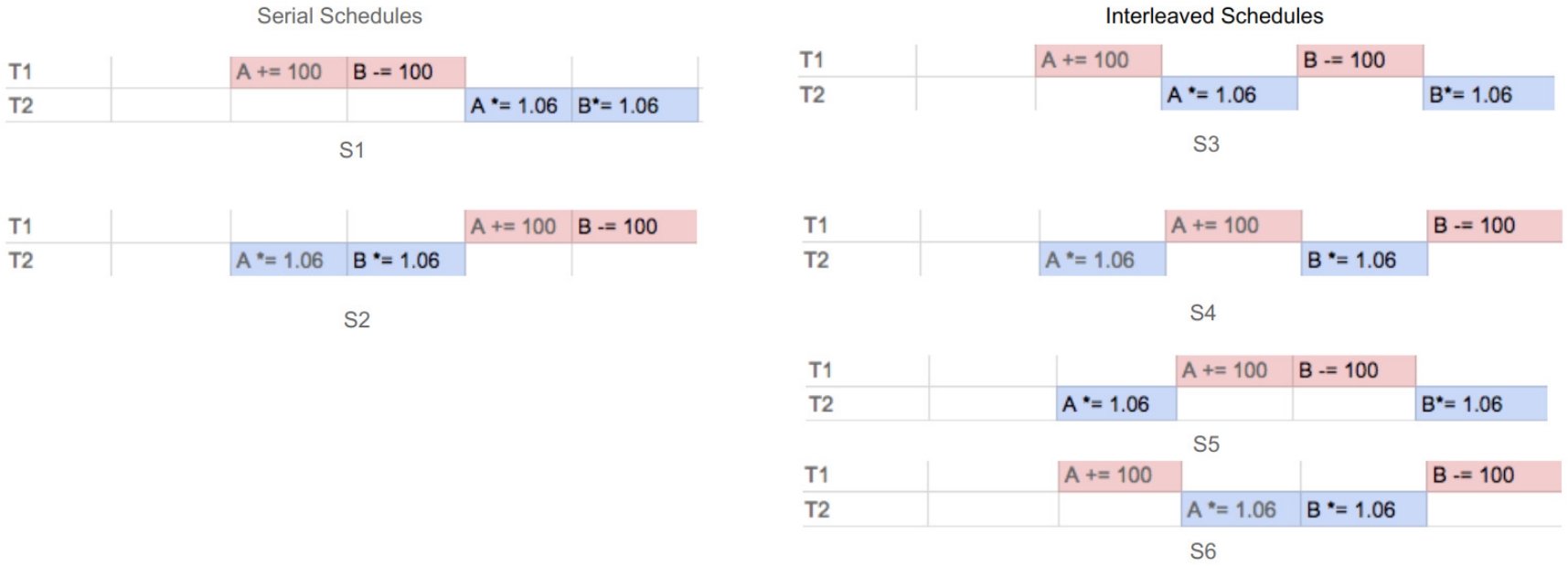
Otras planificaciones (no serie) dejan que las TXN ocurran simultáneamente, intercalando sus operaciones (R/W)
Riesgo: puede dar lugar a datos inconsistentes

SGBD debe garantizar que con el **intercalado/planificación** propuesto se garantizan las propiedades ACID

Un gran poder conlleva una gran responsabilidad



Ejemplos de planificación



Planificación serie	S_1, S_2
Planificación serializable	S_3, S_4
Planificación equivalente	$< S_1, S_3 > < S_2, S_4 >$
Planificación no serializable (:()	$< S_5, S_6 >$

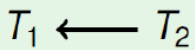
Comprobar si una planificación es serializable

Dada una planificación, se crea un **grafo de precedencia** de la siguiente forma:

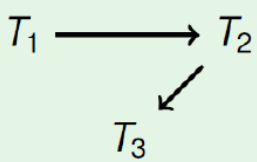
- El grafo tiene un nodo para cada txn
- Habrá una arista de T_1 a T_2 si hay una operación en conflicto entre T_1 y T_2 y T_1 ocurre antes.

T_1	W(V)				W(V)	
T_2			R(V)			

T_1	R(V)				W(V)	
T_2			R(V)			



T_1			W(Y)			
T_2	R(V)				R(Y)	W(Z)
T_3				W(V)		

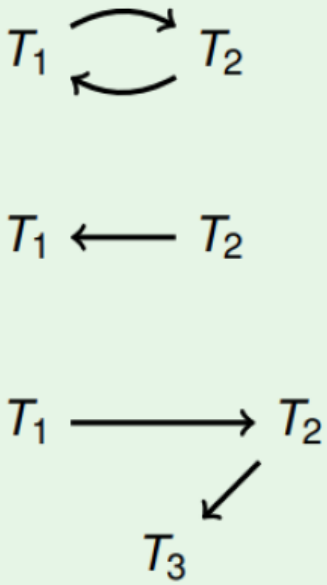


Una planificación es **serializable** si **no hay ciclos** en el grafo de precedencia

T_1	W(V)				W(V)	
T_2			R(V)			

T_1	R(V)				W(V)	
T_2			R(V)			

T_1				W(Y)		
T_2	R(V)				R(Y)	W(Z)
T_3			W(V)			



Schedules 2 and 3 have no cycles in their precedence graph.
They are conflict serializable!

Ejercicio

Considera txn[1] T_1, T_2 y T_3 y dos planificaciones posibles

Planificación 1	Planificación 2
$R_1(X)$	$R_1(X)$
$R_3(X)$	$R_3(X)$
$W_1(X)$	$W_3(X)$
$R_2(X)$	$W_1(X)$
$W_3(X)$	$R_2(X)$

Determina qué planificación, si la hay, es serializable.

[1] Las operaciones W escriben un valor diferente al original

¿Son estas transacciones serializables?

Mira este video `https://www.youtube.com/watch?v=U3SHusK80q0`

Section 4

Anomalías que pueden ocurrir

Garantizar la serializabilidad

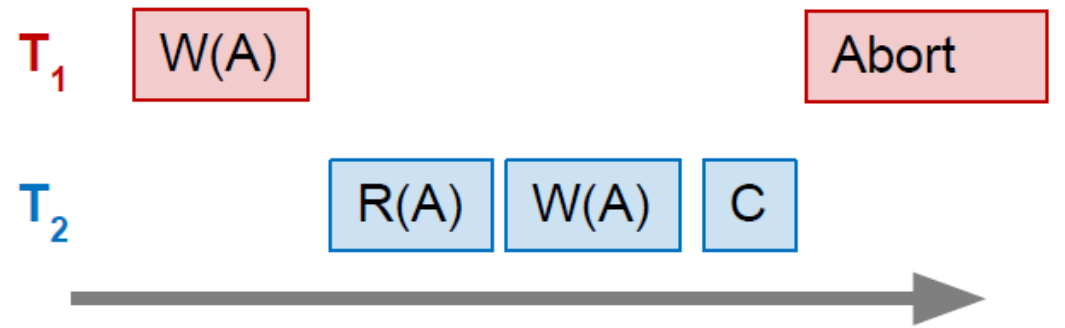
Hasta ahora, hemos visto una condición suficiente que nos permite comprobar si una planificación es serializable.

Pero, ¿cómo garantizar la serialización durante el tiempo de ejecución?

Desafío: el sistema no sabe de antemano qué transacciones ejecutarán y a qué datos accederán.

Dirty read / Lectura sucia / Lectura de datos no comprometidos

Anomalías clásicas con ejecución intercalada

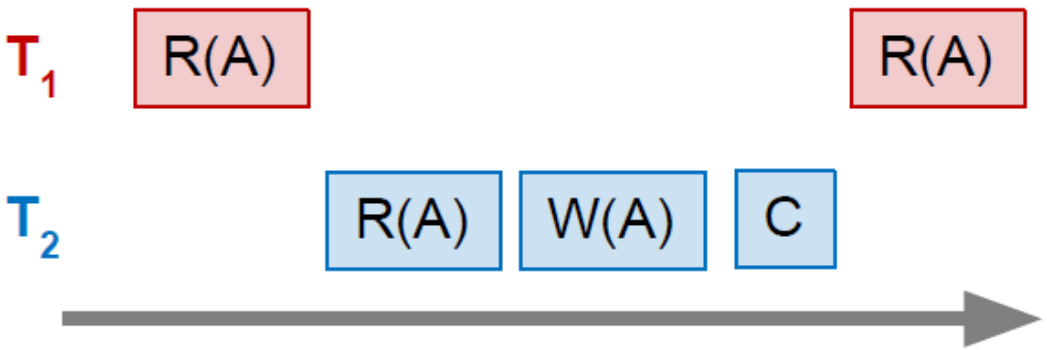


Ocurre debido a un conflicto WR

1. T_1 **escribe** un valor en A
2. T_2 **lee** valor de A, escribe en A, commit
3. T_1 aborta - en este momento, el resultado producido por T_2 está basado en un valor obsoleto / inconsistente

Unrepeatable read, Lectura no repetible

Anomalías clásicas con ejecución intercalada



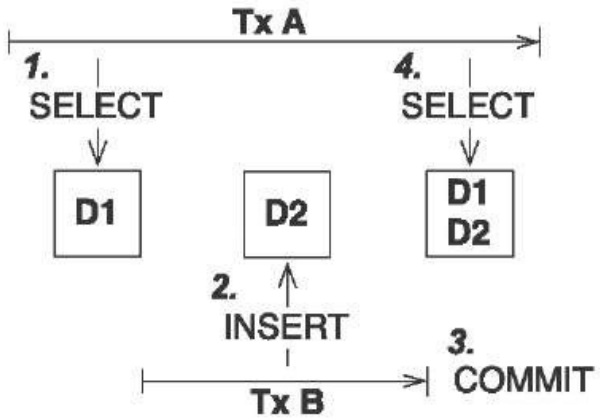
1. T_1 **lee** valor de A
2. T_2 **escribe** valor en A
3. T_1 lee valor de A otra vez y ahora obtiene un valor diferente/ inconsistente

Ocurre debido a un conflicto **RW**

Phantom Read, Lectura fantasma

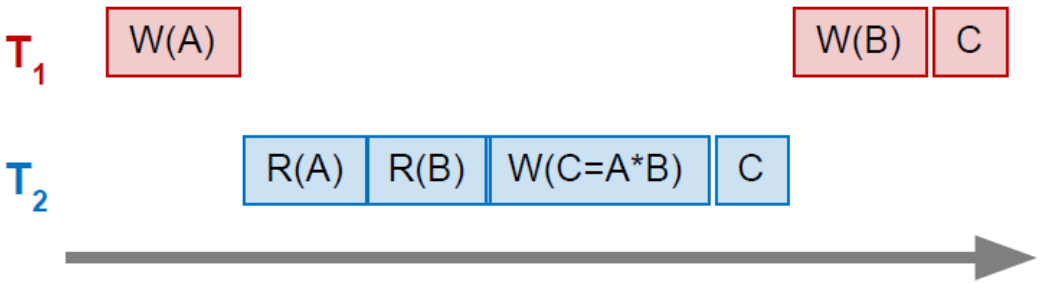
Anomalías clásicas con ejecución intercalada

Phantom Read ocurre cuando se ejecutan dos consultas iguales dentro de una transacción, pero el número de filas recuperadas en ambas es distinto.



Inconsistent read, Lectura inconsistente / Reading partial commits

Anomalías clásicas con ejecución intercalada

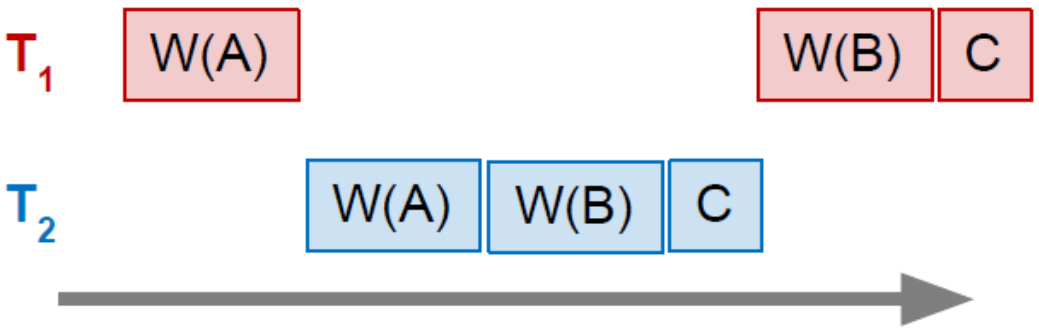


1. T_1 **escribe** un valor en A
2. T_2 **lee** A y B, escribe un valor que depende de A y B
3. T_1 escribe en B - ahora el resultado de T_2 se basa en valores inconsistentes

Nuevamente, ocurre debido a un conflicto WR

Partially-lost update, Pérdida parcial de la actualización

Anomalías clásicas con ejecución intercalada



1. T_1 **escribe** valor en A
2. T_2 **escribe** en A y B
3. T_1 luego escribe en B; ahora el valor de T_1 es el de B y el de T_2 el de A - no equivalente a ninguna planificación serie!

Ocurre debido a un conflicto WW

Ejercicios

Supongamos que el valor inicial de A y B es 1000.
Supón que la T_1 transfiere 50 de A a B y la transacción T_2 retira el 10% del valor de A.

	T_1	T_2	
i_{11}	x = R(A)	y = R(A)	i_{21}
i_{12}	x=x-50	temp=y*0.1	i_{22}
i_{13}	W(A=x)	aux=y-temp	i_{23}
i_{14}	R(y=B)	W(A=aux)	i_{24}
i_{15}	y=y+50	COMMIT	i_{25}
i_{16}	W(B=y)		
i_{17}	COMMIT		

La tabla 1 muestra los valores finales si txn se ejecutan en orden de serie.

If T1 then T2	If T2 then T1
Final values are, A = 855 B = 1050	Final values are, A = 850 B = 1050

Table 1: Final values of A and B if T1 and T2 are executed in serial order

Ahora, considera esta planificación i_{11}, i_{12}, i_{21}, i_{22}, i_{23}, i_{13}, i_{14}, i_{24}, i_{25}, i_{15}, i_{16}, i_{17}. Identificar la anomalía y el conflicto (RW, WR, WW)

Supón que el valor inicial de A y B es 1000; la transacción T_1 transfiere 50 de la cuenta A a la B, y la transacción T_2 calcula el 10% de interés del saldo de A y lo suma al valor de A.

	T_1	T_2	
i_{11}	R(A)	R(A)	i_{21}
i_{12}	A=A-50	aux=A * 0.1	i_{22}
i_{13}	W(A)	A=A+aux	i_{23}
i_{14}	R(B)	W(A)	i_{24}
i_{15}	B=B+50	COMMIT	i_{25}
i_{16}	W(B)		
i_{17}	COMMIT		

Ahora, considera la ejecución $i_{11}, i_{12}, i_{13}, i_{21}, i_{22}, i_{23}, i_{24}, i_{25}$. Mientras T_2 se estaba ejecutando, T_1 decide **rollback** por algún motivo. Identificar la anomalía y el conflicto (RW, WR, WW)

Section 5

¿Cómo gestionar los conflictos ?

Control de concurrencia basado en bloqueos

Enfoque pesimista

Evitar conflictos empleando un mecanismo de **bloqueo**, **pesimista** o **Lock-based Concurrency Control**

- Cada lectura requiere un bloqueo compartido (**shared lock**), mientras que cada escritura requiere un bloqueo exclusivo (**exclusive lock**).
- Un bloqueo compartido no permite la escritura simultánea (bloquea escritores) pero sí que se acceda a los datos para leer (permite lectores concurrentes).
- Un bloqueo exclusivo no permite ni lecturas ni escrituras concurrentes.

Sin embargo, el bloqueo genera competencia por los recursos, y la competencia afecta la escalabilidad.

Schedule with explicit lock actions

T_1				X(B)	W(B)	U(B)			
T_2	S(A)	R(A)					X(B)	W(B)	U(A) U(B)

Here we use the following abbreviations:

- S(A) = shared lock on A
- X(A) = exclusive lock on A
- U(A) = unlock A, or if more precision is needed
  - US(A) = unlock shared lock on A
  - UX(A) = unlock exclusive lock on A

Control de la concurrencia basado en versiones **(Multi-Versioned Concurrency Control (MVCC))**

Enfoque optimista

- Dar por supuesto que nada malo va a suceder
- las txn pueden ejecutar libremente operaciones de lectura/escritura
- solo en el commit, se **detecta** si hay ocurrido algún conflicto

Las transacciones proceden en tres fases

1. **Fase de simulación**: La transacción se ejecuta, pero no escribe los datos a disco. Las actualizaciones se guardan en el espacio de trabajo privado de la transacción.
2. **Fase de validación**: Cuando la transacción quiere confirmar los datos, el SGBD comprueba si su ejecución fue correcta (solo ocurrieron conflictos aceptables). Si no, cancela la transacción.
3. **Fase de escritura**: Actualiza los valores en la BBDD con los del espacio de trabajo privado.

Is this schedule serializable?

T_1	R(X)	W(X)			R(Y)	W(Z)
T_2			R(X)	W(Y)		

No

But what if we had a copy of the old values available?

Then we could do:

Multi-version

T_1	R(X)	W(X)			R(Y-old)	W(Z)
T_2			R(X)	W(Y)		

This is can be serialised to:

T_1	R(X)	W(X)	R(Y)	W(Z)		
T_2					R(X)	W(Y)

Niveles de aislamiento

La aplicación normalmente no establece bloqueos ni captura instantáneas de la BBDD manualmente.

Puede proporcionar al SGBD **sugerencias** para ayudarlo a mejorar la concurrencia: **Niveles de aislamiento**

Un nivel de aislamiento representa una estrategia de bloqueo particular empleada en el sistema de base de datos para mejorar la consistencia de los datos.

O, en MVCC, la forma de implementar lecturas utilizando instantáneas tomadas en un momento específico.

- **Por transacción**
- **El efecto se aplica a las sentencias de lectura**: Establecer un nivel de aislamiento obliga a la txn a comprobar si las **lecturas** cumplen el nivel de aislamiento o no.
- **A los ojos del espectador**: el nivel de aislamiento de una transacción no afectará a otras txn.

1. **Lectura no confirmada (menos restrictiva)**
   - Los datos físicamente corruptos no se leen
   - Incluso se pueden leer registros no confirmados.
2. **Read Committed**
   - No permite la lectura de datos no confirmados (**bloqueo** o **instantánea** antigua).
   - pero las lecturas sucesivas del mismo dato pueden devolver valores diferentes (pero comprometidos).
3. **Repeatable Read**
   - Solo se permiten lecturas de registros comprometidos, las lecturas repetidas del mismo registro deben devolver el mismo valor.
   - Sin embargo, una transacción puede no ser serializable: puede encontrar algunos registros insertados por una transacción pero no encontrar otros.
4. **Serializable (Most Restrictive)**
   - Evita las actualizaciones o la adición de nuevas filas hasta que se complete la transacción.

Anomalías vs Niveles aislamiento

Level/Anomaly	Dirty Read	Unrepetable read	Phantom
Read Uncommitted	Maybe	Maybe	Maybe
Read Committed	No	Maybe	Maybe
Repeatable Read	No	No	Maybe
Serializable	No	No	No

Ejercicios

Sea la tabla `W(name,pay)` y dos txns concurrentes. Supón que pay=50 antes de que se ejecute ninguna instrucción. Recuerda, las instrucciones SQL individuales **siempre se ejecutan atómicamente**.

T_1:
Begin Transaction
S1: update W set pay = 2*pay where name = 'Amy'
S2: update W set pay = 3*pay where name = 'Amy'
Commit

T_2:
Begin Transaction
S3: update W set pay = pay-20 where name = 'Amy'
S4: update W set pay = pay-10 where name = 'Amy'
Commit

Supón que T_1 y T_2 se ejecutan hasta el final, con diferentes niveles de aislamiento. ¿Cuáles son los valores posibles de pay si

1. ambos se ejecutan con Serializable?
2. ambos se ejecutan con Read-Committed?.
3. T_1 es read-committed y T_2 es read-uncommitted?.
4. ambos se ejecutan con Read-Uncommitted?.
5. ambos se ejecutan con nivel de aislamiento Serializable. T_1 se ejecuta hasta el final, pero T_2 aborta después de la instrucción S3 y no se vuelve a ejecutar.

Section 6

Gestión de transacciones JDBC

Las transacciones **no se inician explícitamente**

- Se inician automáticamente justo después de que se crea una conexión o
- inmediatamente después del final de otra transacción.

La conexión tiene un estado llamado `modo AutoCommit`

- `true`, cada sentencia se confirma automáticamente
- `false`, cada sentencia se añade a la txn en curso y debe ser confirmada explícitamente usando `con.commit()` o `con.rollback()`
- Predeterminado: verdadero

Las transacciones se cierran automáticamente, con un commit o rollback explícito, por cualquier fallo o cuando se cierra la conexión.

Interfaz de conexión para gestión de transacciones en JDBC

- **Establece el nivel de aislamiento para la conexión actual**

```
public int getTransactionIsolation() and

void setTransactionIsolation(int level)
```

- Especifica si las transacciones en esta conexión son de solo lectura

```
public boolean getReadOnly() and

void setReadOnly(boolean b)
```

- Si la autocommit no está activado, entonces una transacción

  se confirma usando `connection.commit()` o

  se aborta usando `connection.rollback()`.

- Para verificar el modo de confirmación, use

  ```
  public boolean getAutoCommit()
  ```

  y para cambiarlo,

  ```
  void setAutoCommit(boolean b)
  ```

- Comprobar si la conexión está todavía abierta

  ```
  public boolean isClosed()
  ```

Lecturas adicionales (muy recomendables) (1)

[1] https://www.codemag.com/Article/0607081/Database-Concurrency-Conflicts-in-the-Real-World

[2] https://www.youtube.com/watch?v=FA85kHsJss4 and https://www.youtube.com/watch?v=aNCpEC0-VVs

[3] https://www.youtube.com/playlist?list=PLroEs25KGvwzmvIxYHRhoGTz9w8LeXek0 12-01, 12-02, 12-03

[4] https://cs.stanford.edu/ chrismre/

[5] https://cs.stanford.edu/people/widom/

[6] https://es.slideshare.net/brshristov/database-transactions-and-sql-server-concurrency

Lecturas adicionales (muy recomendables) (2)

[7] https://ucbrise.github.io/cs262a-spring2018/notes/07-concurrency.pdf

[8] https://15445.courses.cs.cmu.edu/fall2018/