Bases de Datos

Clase 16: Transacciones

Hasta ahora

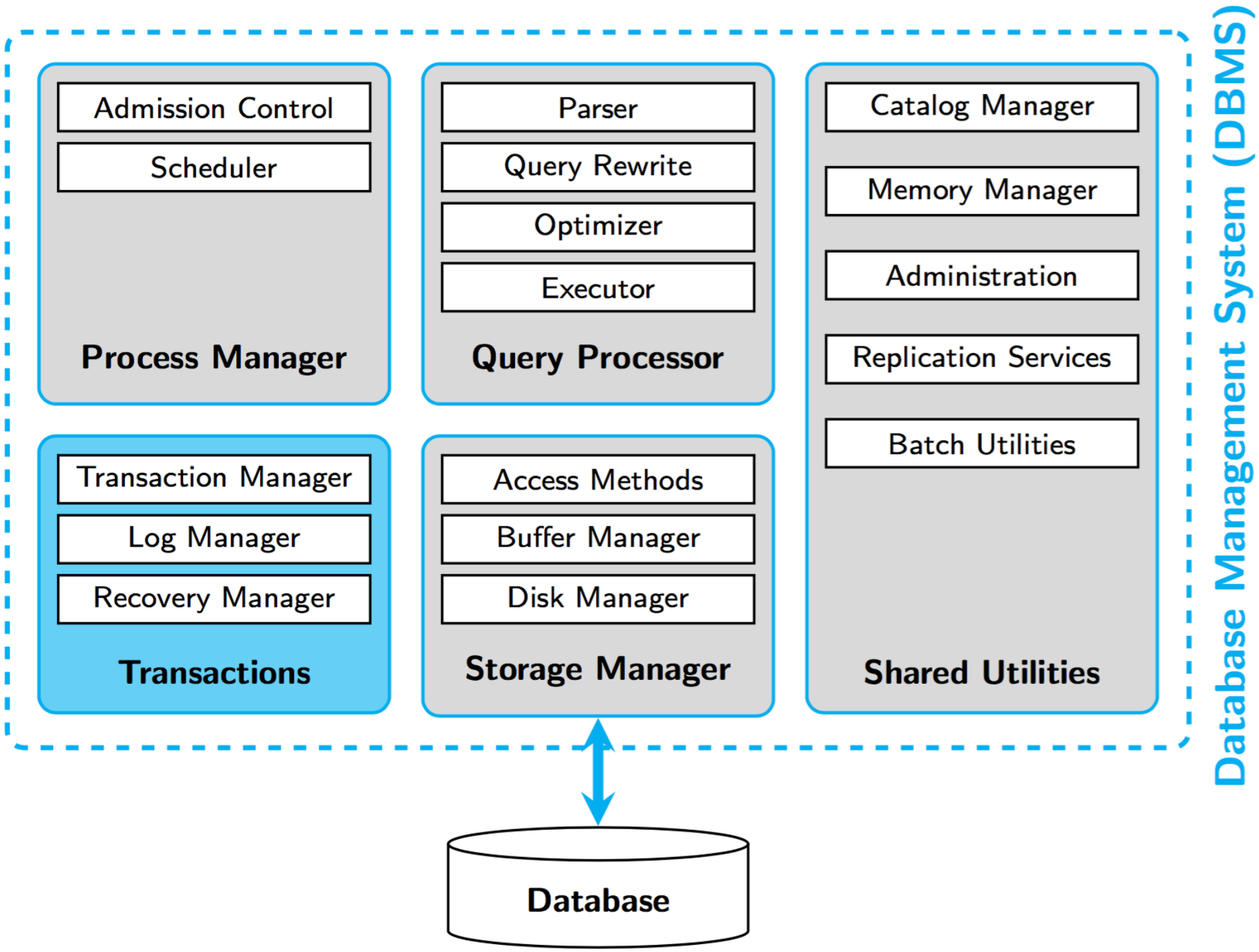
Estamos solos

Hasta ahora

~~Estamos solos~~

No estamos solos

Transactions



Transactions

Componente que asegura las propiedades **ACID**

Transactions

Componente que asegura las propiedades **ACID**

Transactions

Componente que asegura las propiedades **ACID**

Atomicity
Consistency
Isolation
Durability

Transactions

Transaction Manager se encarga de asegurar Isolation y Consistency

Transactions

Transaction Manager se encarga de asegurar Isolation y Consistency

Log y Recovery Manager se encargan de asegurar Atomicity y Durability

Transacciones

Supongamos las siguientes consultas (transferencia de dinero entre dos cuentas):

```
UPDATE cuentas
SET saldo = saldo - v
WHERE cid = 1

UPDATE cuentas
SET saldo = saldo + v
WHERE cid = 2
```

Transacciones

¿Qué pasa cuando el acceso es concurrente?

Transacciones

Transferencia doble

Supongamos que Alice y Bob están casados y tienen una cuenta común

Alice quiere transferirle 100 a su amigo Charles

Bob quiere transferirle 200 a su amigo Charles

Transacciones

Transferencia doble

¿Qué puede salir mal?

Transacciones

Transferencia doble

¿Qué puede salir mal?

Transacciones

Transferencia doble

¿Qué puede salir mal?

Transacciones

Transferencia doble

¿Qué puede salir mal?

Transacciones

Transferencia doble

¿Qué puede salir mal?

Transacciones

Transferencia doble

Proceso Alice	Proceso Bob	Saldo Cuenta A & B	Saldo Cuenta C
`READ(saldoAB, x)`		1000	1000
`WRITE(saldoAB, x - 100)`		900	
`READ(saldoC, x)`			
`WRITE(saldoC, x + 100)`			1100
	`READ(saldoAB, y)`		
	`WRITE(saldoAB, y - 200)`	700	
	`READ(saldoC, y)`		
	`WRITE(saldoC, y + 200)`	700	1300

Transacciones

Transferencia doble

¿Qué puede salir mal?

Transacciones

Transferencia doble

Proceso Alice	Proceso Bob	Saldo Cuenta A & B	Saldo Cuenta C
`READ(saldoAB, x)`		1000	1000
`WRITE(saldoAB, x - 100)`		900	
	`READ(saldoAB, y)`		
	`WRITE(saldoAB, y - 200)`	700	
	`READ(saldoC, y)`		
	`WRITE(saldoC, y + 200)`		1200
`READ(saldoC, x)`			
`WRITE(saldoC, x + 100)`		700	1300

Transacciones

Transferencia doble

¿Qué puede salir mal?

Transacciones

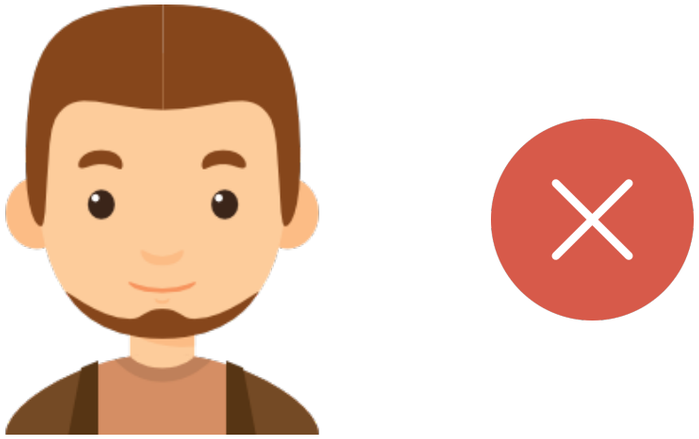
Transferencia doble

Proceso Alice	Proceso Bob	Saldo Cuenta A & B	Saldo Cuenta C
`READ(saldoAB, x)`		1000	1000
`WRITE(saldoAB, x - 100)`		900	
`READ(saldoC, x)`			
	`READ(saldoAB, y)`		
	`WRITE(saldoAB, y - 200)`	700	
	`READ(saldoC, y)`		1200
	`WRITE(saldoC, y + 200)`		1200
`WRITE(saldoC, x + 100)`		700	1100

Transacciones

Transferencia doble

¿Qué puede salir mal?



Transacciones

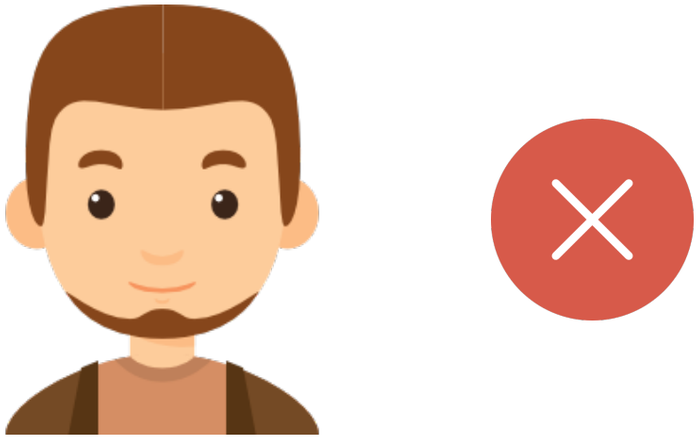
Transferencia doble

Proceso Alice	Proceso Bob	Saldo Cuenta A & B	Saldo Cuenta C
`READ(saldoAB, x)`		1000	1000
`WRITE(saldoAB, x - 100)`		900	
	ERROR	900	1000

Transacciones

Transferencia doble

¿Qué puede salir mal?



Necesitamos transacciones

Una **transacción** es una secuencia de 1 o más operaciones que modifican o consultan la base de datos

Necesitamos transacciones

Una **transacción** es una secuencia de 1 o más operaciones que modifican o consultan la base de datos

- Transferencias de dinero entre cuentas
- Compra por internet
- Registrar un curso
- ...

Transacciones en SQL

```
START TRANSACTION

UPDATE cuentas
SET saldo = saldo - v
WHERE cid = 1

UPDATE cuentas
SET saldo = saldo + v
WHERE cid = 2

COMMIT
```

Transacciones en SQL

`START TRANSACTION` y `COMMIT` nos permiten agrupar operaciones en una sola transacción

Sobre transacciones

- Uno de los componentes fundamentales de una DBMS
- Fundamental para aplicaciones que requieren seguridad
- Uno de los **Turing Award** en Bases de Datos

Turing Award en BD

(Paréntesis)

- 1973 - Charles Bachman, por entregar los primeros cimientos para DBMS

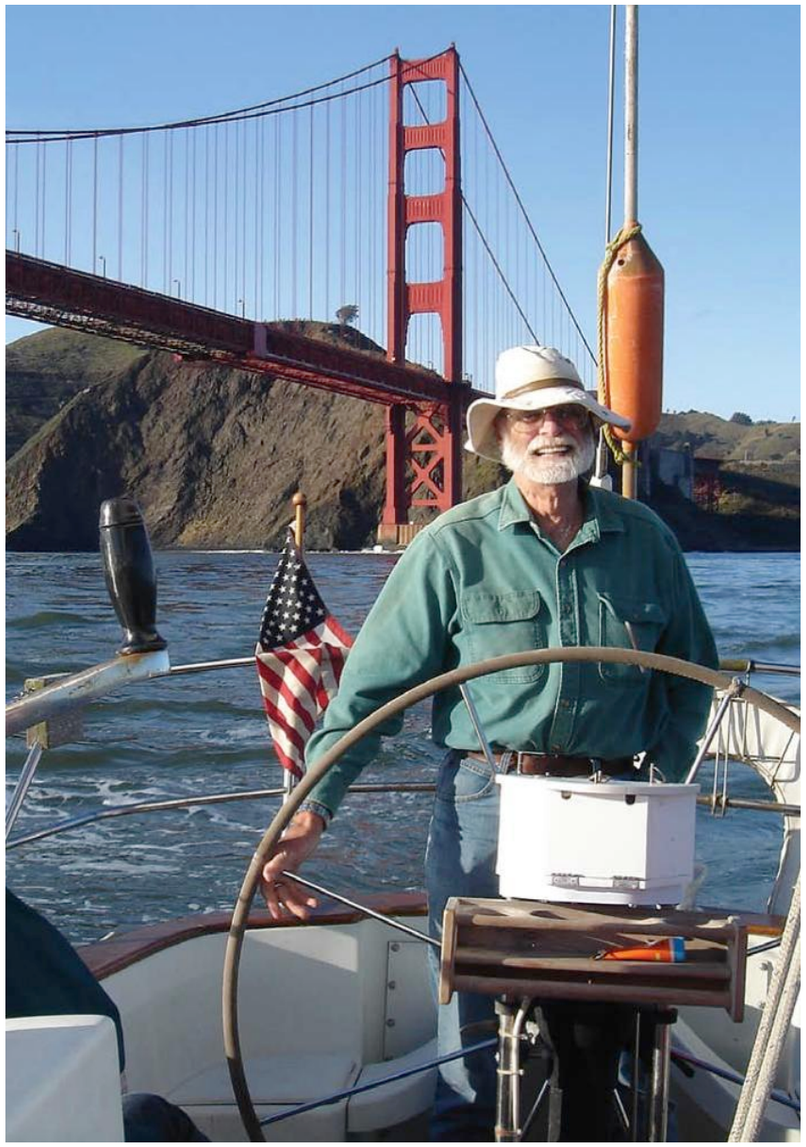
- 1981 - Edgar Codd, por inventar el modelo relacional

- 1998 - Jim Gray, por inventar las **transacciones**

- 2015 - Michael Stonebracker, por desarrollar Ingres

Jim Gray

(Paréntesis)



Conflictos con Transacciones

- Lecturas sucias (Write - Read)
- Lecturas irrepetibles (Read - Write)
- Reescritura de datos temporales (Write - Write)

Conflictos con Transacciones

Lectura sucia

T1	T2	A	B
READ(A,x)		1000	1000
WRITE(A, x - 100)		900	
	READ(A, y)		
	WRITE(A, y * 1.1)	990	
	READ(B, y)		
	WRITE(B, y * 1.1)		1100
READ(B, x)			
WRITE(B, x + 100)		990	1200

Conflictos con Transacciones

Lectura sucia

T1 pudo dejar inconsistente la base de datos, para luego hacerla consistente

T2 pudo leer justo en el momento en que la base de datos estaba inconsistente

Conflictos con Transacciones

Lectura irrepetible

T1	T2	A
READ(A,x)		1
IF(x > 0)		
	READ(A, y)	
	IF(y > 0)	
	WRITE(A, y – 1)	0
	ENDIF	
WRITE(A, x – 1)		-1
ENDIF		-1

Conflictos con Transacciones

Escritura de datos temporales

Imaginemos dos valores que siempre tienen que ser iguales

Conflictos con Transacciones

Escritura de datos temporales

T1	T2	A	B
WRITE(A,10)		10	
WRITE(B,10)			10
	WRITE(A,20)	20	
	WRITE(B,20)	20	20

Conflictos con Transacciones

Escritura de datos temporales

T1	T2	A	B
WRITE(A,10)		10	
	WRITE(A,20)	20	
	WRITE(B,20)		20
	WRITE(B,10)	20	10

Schedule

Un **schedule S** es una secuencia de operaciones primitivas de una o más transacciones, tal que para toda transacción, las acciones de ella aparecen en el mismo orden que en su definición

Schedule

Transacciones de un schedule

T1	T2
READ(A,x)	
	READ(A,y)
x:= x + 100	
	y:= y * 2
WRITE(A,x)	
	WRITE(A,y)
READ(B,x)	
	READ(B,y)
x:= x + 200	
	y:= y * 3
WRITE(B,x)	
	WRITE(B,y)

Schedule

Un schedule

T1	T2
READ(A,x)	
x:= x + 100	
WRITE(A,x)	
READ(B,x)	
x:= x + 200	
WRITE(B,x)	
	READ(A,y)
	y:= y * 2
	WRITE(A,y)
	READ(B,y)
	y:= y * 3
	WRITE(B,y)

Schedule

Otro schedule

T1	T2
READ(A,x)	
x:= x + 100	
WRITE(A,x)	
	READ(A,y)
	y:= y * 2
	WRITE(A,y)
READ(B,x)	
x:= x + 200	
WRITE(B,x)	
	READ(B,y)
	y:= y * 3
	WRITE(B,y)

Schedule Serial

Un **schedule S** es **serial** si no hay intercalación entre las acciones

Schedule Serial

Un schedule serial

T1	T2
READ(A,x)	
x:= x + 100	
WRITE(A,x)	
READ(B,x)	
x:= x + 200	
WRITE(B,x)	
	READ(A,y)
	y:= y * 2
	WRITE(A,y)
	READ(B,y)
	y:= y * 3
	WRITE(B,y)

Schedule Serializable

Un **schedule S** es **serializable** si existe algún **schedule S'** serial con las mismas transacciones, tal que el resultado de **S** y **S'** es el mismo para todo estado inicial de la BD

Schedule Serializable

T1	T2
READ(A,x)	
x:= x + 100	
WRITE(A,x)	
	READ(A,y)
	y:= y * 2
	WRITE(A,y)
READ(B,x)	
x:= x + 200	
WRITE(B,x)	
	READ(B,y)
	y:= y * 3
	WRITE(B,y)

Schedule No Serializable

T1	T2
READ(A,x)	
x:= x + 100	
WRITE(A,x)	
	READ(A,y)
	y:= y * 2
	WRITE(A,y)
	READ(B,y)
	y:= y * 3
	WRITE(B,y)
READ(B,x)	
x:= x + 200	
WRITE(B,x)	

Transacciones

La tarea del Transaction Manager es permitir solo schedules que sean **serializables**

¿Cómo determinamos de manera rápida si un schedule es serializable?

Transacciones

Notación

Si la transacción `i` ejecuta `READ(X,t)` escribimos `Ri(X)`

Si la transacción `i` ejecuta `WRITE(X,t)` escribimos `Wi(X)`

Acciones No Conflictivas

Las siguientes acciones son NO conflictivas para dos transacciones distintas `i, j`:

- `Ri(X), Rj(Y)`
- `Ri(X), Wj(Y) con X != Y`
- `Wi(X), Rj(Y) con X != Y`
- `Wi(X), Wj(Y) con X != Y`

Podemos cambiarlas de orden en un **schedule**!

Acciones Conflictivas

Las siguientes acciones son conflictivas para dos transacciones distintas `i, j`:

- `Pi(X), Qi(Y)` con `P,Q` en `{R, W}`
- `Ri(X), Wj(X)`
- `Wi(X), Rj(X)`
- `Wi(X), Wj(X)`

No podemos cambiar su orden en un **schedule** a la ligera!

Acciones Conflictivas

Puedo permutar un par de operaciones consecutivas si:

- No usan el mismo recurso
- Usan el mismo recurso pero ambas son de lectura

Un **schedule** es *conflict serializable* si puedo transformarlo a uno **serial** usando permutaciones.

Acciones Conflictivas

Si un **schedule** es *conflict serializable* implica que también es serializable, pero hay schedules serializables que no son *conflict serializable*

Grafo de precedencia

Dado un **schedule** puedo construir su grafo de precedencia

- Nodos: **transacciones** del sistema
- Aristas: hay una arista de **T** a **T'** si **T** ejecuta una operación `op1` antes de una operación `op2` de **T'**, tal que `op1` y `op2` no se pueden permutar

Grafo de precedencia

Teorema Un schedule es *conflict serializable* ssi el grafo de precedencia es acíclico

Además, determinar si un *schedule* es serializable es NP-Completo!

Grafo de precedencia

Ejemplo (Pizarra)

¿Es serializable?

T1	T2	T3
	R2(A)	
R1(B)		
	W2(A)	
		R3(A)
W1(B)		
		W3(A)
	R2(B)	
	W2(B)	

Grafo de precedencia

Ejemplo (Pizarra)

¿Es *conflict serializable*?

T1	T2	T3
	R2(A)	
R1(B)		
	W2(A)	
	R2(B)	
		R3(A)
W1(B)		
		W3(A)
	W2(B)	

Strict 2PL

Es el protocolo para control de concurrencia más usado en los DBMS

Está basado en la utilización de locks

Tiene dos reglas

Strict 2PL

Regla 1

Si una transacción T quiere leer (resp. modificar) un objeto, primero pide un **shared lock** (resp. **exclusive lock**) sobre el objeto

Una transacción que pide un lock se suspende hasta que el lock es otorgado

Strict 2PL

Regla 1

Si una transacción mantiene un exclusive lock de un objeto, ninguna otra transacción puede mantener un shared o exclusive lock sobre el objeto

Es importante notar que por lo anterior, para obtener el exclusive lock, no debe haber ningún lock sobre el objeto

Strict 2PL

Regla 2

Cuando la transacción se completa, libera todos los locks que mantenía

Strict 2PL

Strict 2PL.

Strict 2PL

Estas reglas aseguran solo **schedules** serializables