Técnicas para Big Data

Clase 03 - Propiedades ACID

Hasta ahora

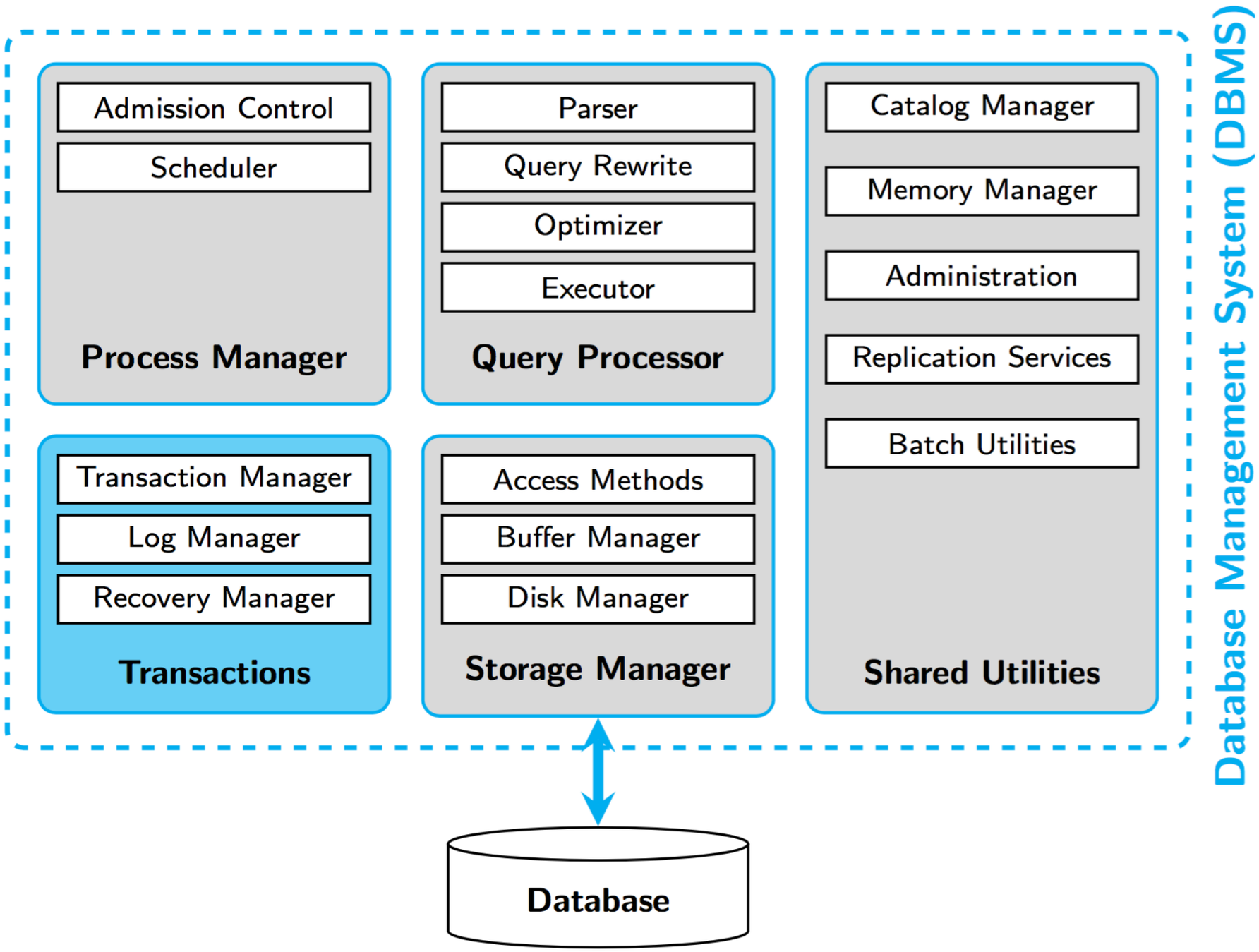
Estamos solos

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

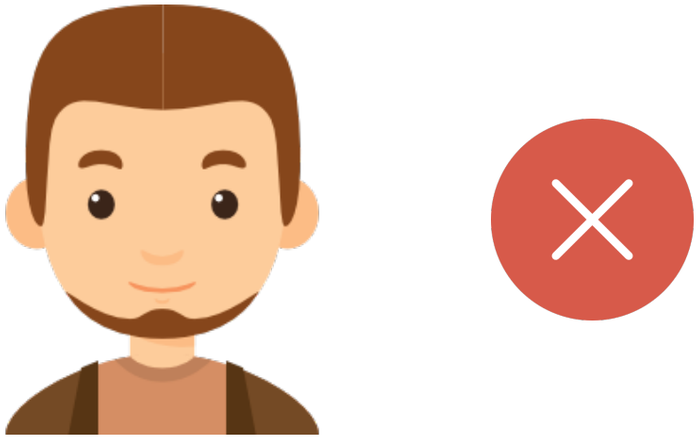
Transferencia doble

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

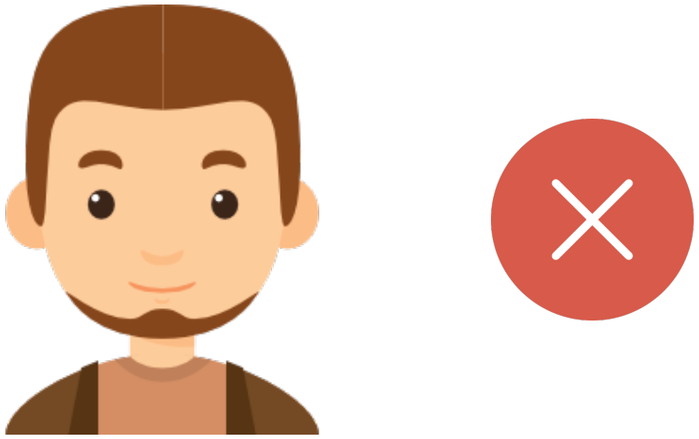
Transferencia doble

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

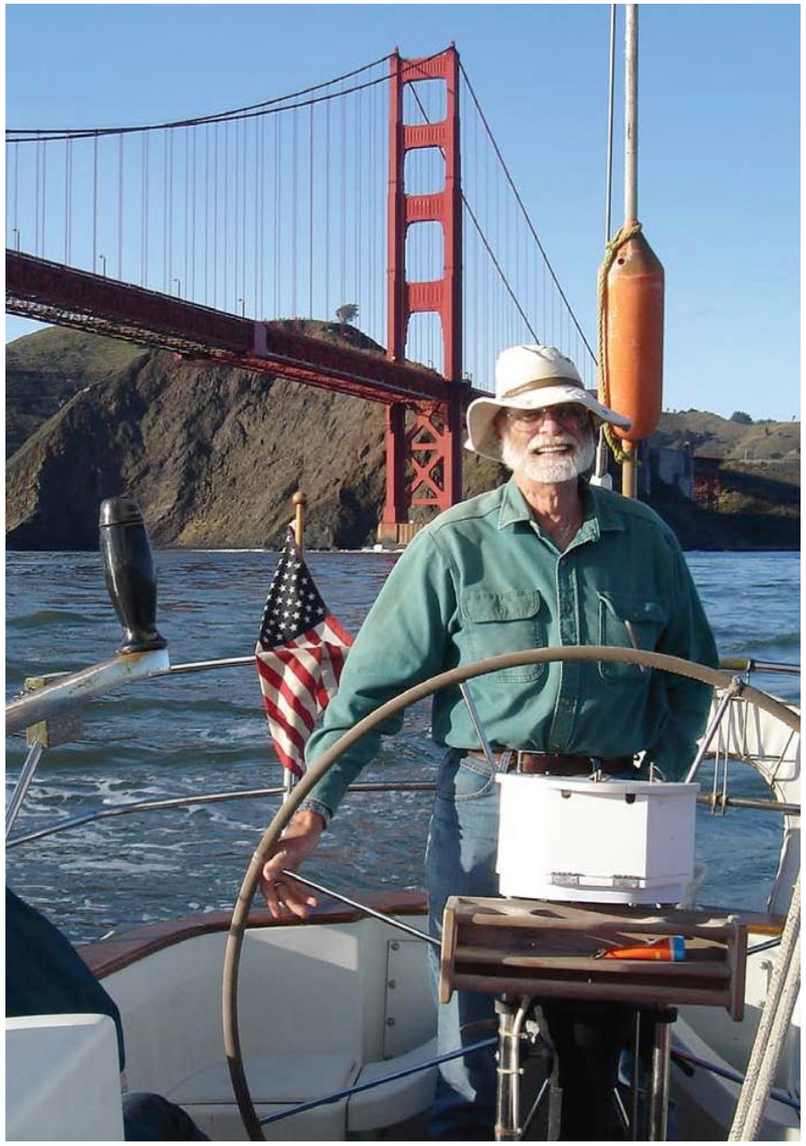
- 1981 - Edgar Codd, por inventar el modelo relacional

- 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

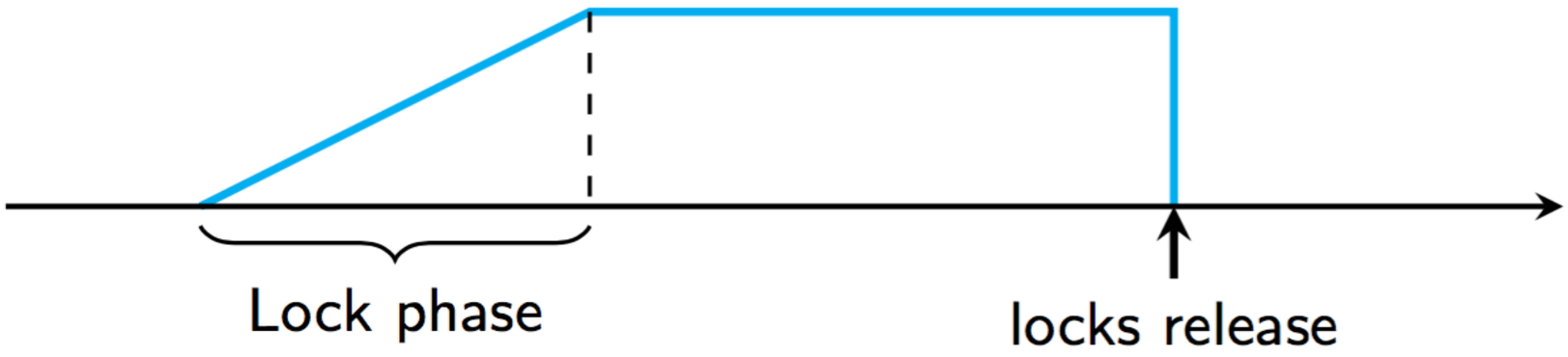
Strict 2PL

Regla 2

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

Strict 2PL

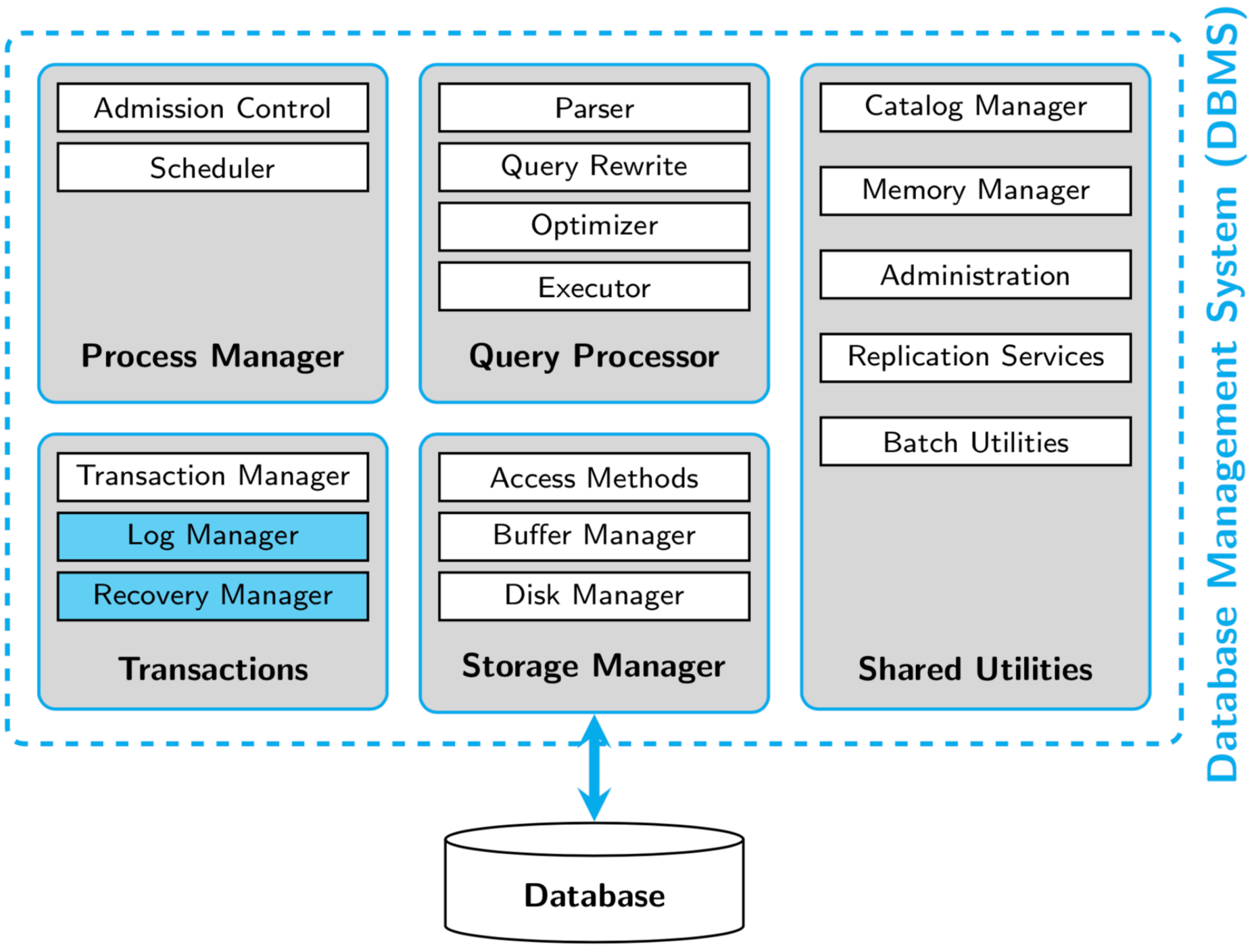
Strict 2PL.



Strict 2PL

Estas reglas aseguran solo **schedules** serializables

Recuperación de Fallas



Recuperación de Fallas

Recuperación de Fallas

Log y Recovery Manager se encargan de asegurar Atomicity y Durability

¿Pero qué puede salir mal?

Fallas en la ejecución:

- Datos erróneos
  - Solución: restricciones de integridad, data cleaning

- Fallas en el disco duro
  - Solución: RAID, copias redundantes

¿Pero qué puede salir mal?

Fallas en la ejecución:

- Catástrofes
  - Solución: copias distribuidas

- Fallas del sistema
  - Solución: **Log y Recovery Manager**

Log Manager

Una página se va llenando secuencialmente con *logs*

Cuando la página se llena, se almacena en disco

Todas las transacciones escriben el *log* de manera concurrente

Log Manager

Registra todas las acciones de las transacciones

Log Records

Los *logs* comunes son:

- <START **T**>
- <COMMIT **T**>
- <ABORT **T**>
- <**T** UPDATE>

¿Cómo los usamos?

Undo Logging

Forma de escribir los *logs* para poder hacer *recovery* del sistema

Undo Logging

Los *logs* son:

- <START **T**>
- <COMMIT **T**>
- <ABORT **T**>
- <**T**, X, t> donde t es el valor **antiguo** de X

Undo Logging

Regla 1: si **T** modifica X, el *log* <**T**, X, `t`> debe ser escrito antes que el valor X sea escrito en disco

Regla 2: si **T** hace *commit*, el log <`COMMIT` **T**> debe ser escrito justo después de que todos los datos modificados por **T** estén almacenados en disco

Undo Logging

En resumen:

- Escribir el log <**T**, X, t>
- Escribir los datos a disco
- Escribir <**COMMIT T**>
- Hacer *flush* a disco del *log*

Recuperación con Undo Logging

Detectando fallas en el *log*:

```
... <START T> ... <COMMIT T> ...
```

Recuperación con Undo Logging

Detectando fallas en el *log*:

```
... <START T> ... <ABORT T> ...
```

Recuperación con Undo Logging

Detectando fallas en el *log*:

... <START T> ...

Recuperación con Undo Logging

Supongamos que mientras usamos nuestro sistema, se apagó de forma imprevista

Leyendo el *log* podemos hacer que la base de datos quede en un estado consistente

Recovery

Algoritmo para un *Undo Logging*

Procesamos el log desde el final hasta el principio:

- Si leo <**COMMIT T**>, marco **T** como realizada
- Si leo <**ABORT T**>, marco **T** como realizada
- Si leo <**T**, X, t>, debo restituir X := t en disco, si no fue realizada.
- Si leo <**START T**>, lo ignoro

Recovery

Algoritmo para un *Undo Logging*

- ¿Hasta dónde tenemos que leer el *log*?
- ¿Qué pasa si el sistema falla en plena recuperación?
- ¿Cómo trucamos el *log*?

Recovery

Uso de *Checkpoints*

Utilizamos *checkpoints* para no tener que leer el *log* entero y para manejar las fallas mientras se hacer *recovery*

Recovery

Uso de *Checkpoints*

- Dejamos de escribir transacciones
- Esperamos a que las transacciones actuales terminen
- Se guarda el *log* en disco
- Escribimos <**CKPT**> y se guarda en disco
- Se reanudan las transacciones

Recovery

Uso de *Checkpoints*

Ahora hacemos *recovery* hasta leer un <CKPT>

Recovery

Uso de *Checkpoints*

Ahora hacemos *recovery* hasta leer un <CKPT>

Problema: es prácticamente necesario apagar el sistema para guardar un *checkpoint*

Recovery

Uso de *Nonquiescent Checkpoints*

Nonquiescent Checkpoints son un tipo de *checkpoint* que no requiere "apagar" el sistema

Recovery

Uso de *Nonquiescent Checkpoints*

- Escribimos un *log* <**START CKPT** (T_1, ..., T_n)>, donde T_1, ..., T_n son transacciones activas

- Esperamos hasta que T_1, ..., T_n terminen, sin restringir nuevas transacciones

- Cuando T_1, ..., T_n hayan terminado, escribimos <**END CKPT**>

Undo Recovery

Uso de *Nonquiescent Checkpoints*

- Avanzamos desde el final al inicio
- Si encontramos un <**END CKPT**>, hacemos *undo* de todo lo que haya después del inicio del *checkpoint*
- Si encontramos un <**START CKPT** (T_1, ..., T_n)> sin su <**END CKPT**>, debemos analizar el log desde el inicio de la transacción más antigua entre T_1, ..., T_n

Ejemplo

Uso de *Checkpoints* en *Undo Logging*

Considere este *log* después de una falla:

Log
<START T1>
<T1, a, 5>
<START T2>
<T2, b, 10>
<START CKPT (T1, T2)>
<T2, c, 15>
<START T3>
<T1, d, 20>
<COMMIT T1>
<T3, e, 25>
<COMMIT T2>
<END CKPT>

Ejemplo

Uso de *Checkpoints* en *Undo Logging*

Ahora considere este *log* después de una falla:

Log
<START T1>
<T1, a, 5>
<START T2>
<T2, b, 10>
<START CKPT (T1, T2)>
<T2, c, 15>
<START T3>
<T1, d, 20>
<COMMIT T1>
<T3, e, 25>

Undo Logging

Problema: no es posible hacer `COMMIT` antes de almacenar los datos en disco

Por lo tanto las transacciones se toman más tiempo en terminar!

Redo Logging

Los *logs* son:

- <START **T**>
- <COMMIT **T**>
- <ABORT **T**>
- <**T**, X, **v**> donde **v** es el valor **nuevo** de X

Redo Logging

Regla 1: Antes de modificar cualquier elemento **X** en disco, es necesario que todos los *logs* estén almacenados en disco, incluido el **`COMMIT`**

Redo Logging

Regla 1: Antes de modificar cualquier elemento **X** en disco, es necesario que todos los *logs* estén almacenados en disco, incluido el **`COMMIT`**

Esto es al revés respecto a *Undo Logging*

Redo Logging

En resumen:

- Escribir el log <**T**, X, **v**>
- Escribir <**COMMIT T**>
- Hacer *flush* a disco del *log*
- Escribir los datos en disco

Recuperación con Redo Logging

Detectando fallas en el *log*:

```
... <START T> ... <COMMIT T> ...
```

Recuperación con Redo Logging

Detectando fallas en el *log*:

```
... <START T> ... <ABORT T> ...
```

Recuperación con Redo Logging

Detectando fallas en el *log*:

```
... <START T> ...
```

Recovery

Algoritmo para un *Redo Logging*

Procesamos el *log* desde el principo hasta el final:

- Identificamos las transacciones que hicieron `COMMIT`
- Hacemos un *scan* desde el principio
- Si leo <**T**, X, **v**>:
  - Si **T** no hizo `COMMIT`, no hacer nada
  - Si **T** hizo `COMMIT`, reescribir con el valor **v**
- Para cada transacción incompleta, escribir <`ABORT` **T**>

Recovery

Uso de *Checkpoints* en *Redo Logging*

¿Cómo utilizamos los checkpoints en el Redo Logging?

Recovery

Uso de *Checkpoints* en *Redo Logging*

- Escribimos un *log* <**START CKPT** (T_1, ..., T_n)>, donde T_1, ..., T_n son transacciones activas y sin **COMMIT**

- Guardar en disco todo lo que haya hecho **COMMIT** hasta ese punto

- Una vez hecho, escribir <**END CKPT**>

Redo Recovery

Uso de *Checkpoints* en *Redo Logging*

- Revisar el *log* desde el final al inicio
- Si encontramos un <**END CKPT**>, debemos retroceder hasta su su respectivo <**START CKPT** (**T**$_1$, ..., **T**$_n$)>, y comenzar a hacer *redo* desde la transacción más antigua entre **T**$_1$, ..., **T**$_n$
- No se hace *redo* de las transacciones con **COMMIT** antes del <**START CKPT** (**T**$_1$, ..., **T**$_n$)>

Redo Recovery

Uso de *Checkpoints* en *Redo Logging*

Si encontramos un <**START CKPT** (T_1, ..., T_n)> sin su <**END CKPT**>, debemos retroceder hasta encontrar un <**END CKPT**>

Ejemplo

Uso de *Checkpoints* en *Redo Logging*

Considere este *log* después de una falla:

Log
<START T1>
<T1, a, 5>
<START T2>
<COMMIT T1>
<T2, b, 10>
<START CKPT (T2)>
<T2, c, 15>
<START T3>
<T3, e, 25>
<END CKPT>
<COMMIT T2>
<COMMIT T3>

Redo Logging

Problema: no es posible ir grabando los valores de X en disco antes que termine la transacción

Por lo tanto se congestiona la escritura en disco!

Undo/Redo Logging

Es la solución para obtener mayor performance que mezcla las estrategias anteriormente planteadas

Técnicas de Logging

Resumen

	Undo	**Redo**
Trans. Incompletas	Cancelarlas	Ignorarlas
Trans. Comiteadas	Ignorarlas	Repetirlas
Escribir **`COMMIT`**	Después de almacenar en disco	Antes de almacenar en disco
`UPDATE` *Log Record*	Valores antiguos	Valores nuevos

Técnicas para Big Data

Clase 03 - Propiedades ACID