

UNIVERSIDAD NACIONAL DE INGENIERÍA

Bases de Datos

Transacciones: ACID, serializabilidad y aislamiento

Transacciones y sistemas distribuidos

Gerson Garrido

Semana 13

Hoy

Sesión A: ACID y serializabilidad

1. Transacciones y ACID
2. El modelo simple; estados de una transacción
3. Concurrencia y schedules
4. Conflict serializability y el grafo de precedencia
5. View serializability, en breve

Sesión B: recuperabilidad y aislamiento

6. Schedules recuperables y cascadeless
7. Niveles de aislamiento y anomalías
8. Lo que hace PostgreSQL, con dos `psql`
9. Transacciones en SQL
10. Reintentar: `40001` y `40P01`

Lectura

Silberschatz, secs. 17.1–17.6 (sesión A) y 17.7–17.10 (sesión B). CS186, *Transactions & Concurrency I*. PostgreSQL, *Transaction Isolation*.

SECCIÓN 01

Transacciones

01

Registrar un envío

Una agencia recibe un paquete. La aplicación:

1. inserta la fila en `shipment`;
2. inserta el evento `received` en `shipment_event`;
3. le asigna el `trip_id` del viaje.

Registrar un envío

Una agencia recibe un paquete. La aplicación:

1. inserta la fila en `shipment`;
2. inserta el evento `received` en `shipment_event`;
3. le asigna el `trip_id` del viaje.

Para la empresa es una sola operación. No puede pasar:

- que se haga una parte y no el resto (una caída entre 2 y 3);
- que otro empleado vea el envío a medias.

Registrar un envío

Una agencia recibe un paquete. La aplicación:

1. inserta la fila en `shipment`;
2. inserta el evento `received` en `shipment_event`;
3. le asigna el `trip_id` del viaje.

Para la empresa es una sola operación. No puede pasar:

- que se haga una parte y no el resto (una caída entre 2 y 3);
- que otro empleado vea el envío a medias.

Transacción *(transaction)*

Una unidad de ejecución de un programa que accede a datos y posiblemente los actualiza. En SQL, lo que va entre `BEGIN` y `COMMIT`.

Hasta ahora

Semanas 9–12: una consulta a la vez y una computadora que nunca se apaga. Ya no.

ACID

- **Atomicidad:** se reflejan todas las operaciones o ninguna.

ACID

- **Atomicidad:** se reflejan todas las operaciones o ninguna.
- **Consistencia:** ejecutada sola, lleva la base de un estado correcto a otro correcto (restricciones *y* reglas del negocio no declaradas).

ACID

- **Atomicidad:** se reflejan todas las operaciones o ninguna.
- **Consistencia:** ejecutada sola, lleva la base de un estado correcto a otro correcto (restricciones *y* reglas del negocio no declaradas).
- **Aislamiento:** a cada Ti le parece que cada Tj terminó antes de que ella empezara o empezó después de que ella terminó.

ACID

- **Atomicidad:** se reflejan todas las operaciones o ninguna.
- **Consistencia:** ejecutada sola, lleva la base de un estado correcto a otro correcto (restricciones *y* reglas del negocio no declaradas).
- **Aislamiento:** a cada Ti le parece que cada Tj terminó antes de que ella empezara o empezó después de que ella terminó.
- **Durabilidad:** lo que hizo commit sobrevive a cualquier fallo posterior.

ACID

- **Atomicidad:** se reflejan todas las operaciones o ninguna.
- **Consistencia:** ejecutada sola, lleva la base de un estado correcto a otro correcto (restricciones *y* reglas del negocio no declaradas).
- **Aislamiento:** a cada Ti le parece que cada Tj terminó antes de que ella empezara o empezó después de que ella terminó.
- **Durabilidad:** lo que hizo commit sobrevive a cualquier fallo posterior.

Propiedad	Quién la garantiza	Semana
Atomicidad	recuperación, con el log	15
Consistencia	el programador; las restricciones ayudan	4
Aislamiento	control de concurrencia (locks, MVCC)	14
Durabilidad	recuperación, con el log en almacenamiento estable	15

El modelo simple

- La base es un conjunto de **ítems**: A, B, Q.
- `read(X)`: del ítem X a una variable local X.
- `write(X)`: de la variable local al ítem X (en el buffer pool, no en el disco).
- Lo que se calcula en medio, el DBMS no lo ve.

T1: 5 soles de 501 a 502

```
T1:  read(A);  A := A - 5;  write(A)
     read(B);  B := B + 5;  write(B)
```

El modelo simple

- La base es un conjunto de **ítems**: A, B, Q.
- `read(X)`: del ítem X a una variable local X.
- `write(X)`: de la variable local al ítem X (en el buffer pool, no en el disco).
- Lo que se calcula en medio, el DBMS no lo ve.

T1: 5 soles de 501 a 502

```
T1:  read(A);  A := A - 5;  write(A)
     read(B);  B := B + 5;  write(B)
```

- **C:** A + B no cambia. Ninguna restricción lo dice: lo cumple el programa.

El modelo simple

- La base es un conjunto de **ítems**: A, B, Q.
- `read(X)`: del ítem X a una variable local X.
- `write(X)`: de la variable local al ítem X (en el buffer pool, no en el disco).
- Lo que se calcula en medio, el DBMS no lo ve.

T1: 5 soles de 501 a 502

```
T1:  read(A);  A := A - 5;  write(A)
     read(B);  B := B + 5;  write(B)
```

- **C:** A + B no cambia. Ninguna restricción lo dice: lo cumple el programa.
- **A:** si falla entre `write(A)` y `write(B)`, faltan 5 soles. Ese estado existe un rato; no puede quedar visible después del fallo.

El modelo simple

- La base es un conjunto de **ítems**: A, B, Q.
- `read(X)`: del ítem X a una variable local X.
- `write(X)`: de la variable local al ítem X (en el buffer pool, no en el disco).
- Lo que se calcula en medio, el DBMS no lo ve.

T1: 5 soles de 501 a 502

```
T1:  read(A);  A := A - 5;  write(A)
     read(B);  B := B + 5;  write(B)
```

- **C:** A + B no cambia. Ninguna restricción lo dice: lo cumple el programa.
- **A:** si falla entre `write(A)` y `write(B)`, faltan 5 soles. Ese estado existe un rato; no puede quedar visible después del fallo.
- **D:** con commit confirmado, los dos `write` sobreviven.

El modelo simple

- La base es un conjunto de **ítems**: A, B, Q.
- `read(X)`: del ítem X a una variable local X.
- `write(X)`: de la variable local al ítem X (en el buffer pool, no en el disco).
- Lo que se calcula en medio, el DBMS no lo ve.

```
T1: 5 soles de 501 a 502

T1:  read(A);  A := A - 5;  write(A)
     read(B);  B := B + 5;  write(B)
```

- **C:** A + B no cambia. Ninguna restricción lo dice: lo cumple el programa.
- **A:** si falla entre `write(A)` y `write(B)`, faltan 5 soles. Ese estado existe un rato; no puede quedar visible después del fallo.
- **D:** con commit confirmado, los dos `write` sobreviven.
- **I:** otra transacción que suma A + B entre los dos `write` ve 5 de menos.

Almacenamiento volátil, no volátil y estable

- **Volátil:** memoria principal, cachés. No sobrevive a una caída del sistema.
- **No volátil:** disco, SSD, cinta. Sobrevive a la caída; no a la falla del dispositivo.
- **Estable** *(stable storage)*: la información nunca se pierde.

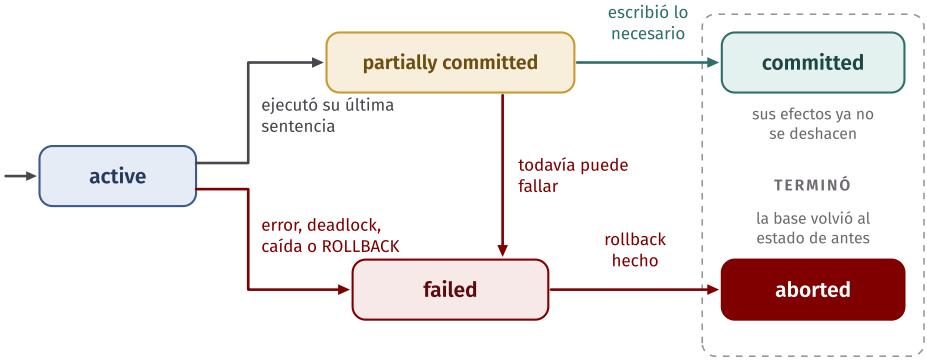
Almacenamiento volátil, no volátil y estable

- **Volátil:** memoria principal, cachés. No sobrevive a una caída del sistema.
- **No volátil:** disco, SSD, cinta. Sobrevive a la caída; no a la falla del dispositivo.
- **Estable** *(stable storage)*: la información nunca se pierde.

Una idealización

No existe un dispositivo estable. Se aproxima con varias copias en medios no volátiles con fallas independientes: varios discos (RAID), mejor aún varios sitios. El log de la semana 15 vive ahí; la durabilidad es tan buena como esa aproximación.

Estados de una transacción



Estados de una transacción: qué significa cada uno

- **partially committed**: ejecutó su última sentencia; aún puede fallar.
- **failed**: no puede seguir (restricción, deadlock, caída, `ROLLBACK`).
- **aborted**: rollback hecho; se reinicia o se mata.

Estados de una transacción: qué significa cada uno

- **partially committed**: ejecutó su última sentencia; aún puede fallar.
- **failed**: no puede seguir (restricción, deadlock, caída, `ROLLBACK`).
- **aborted**: rollback hecho; se reinicia o se mata.

Commit es definitivo

Lo confirmado no se deshace: se corrige con otra transacción, de compensación (un reembolso).

Estados de una transacción: qué significa cada uno

- **partially committed**: ejecutó su última sentencia; aún puede fallar.
- **failed**: no puede seguir (restricción, deadlock, caída, `ROLLBACK`).
- **aborted**: rollback hecho; se reinicia o se mata.

Commit es definitivo

Lo confirmado no se deshace: se corrige con otra transacción, de compensación (un reembolso).

Error común

Mandar el SMS *dentro* de la transacción. Si aborta, la base vuelve atrás; el SMS no. Las escrituras externas van después del commit.

SECCIÓN 02

Concurrencia y schedules

02



¿Por qué no una transacción después de otra?

- **Throughput:** mientras una espera al disco, otra usa la CPU.
- **Tiempo de espera:** un reporte de segundos no debe frenar el registro de un paquete.

¿Por qué no una transacción después de otra?

- **Throughput:** mientras una espera al disco, otra usa la CPU.
- **Tiempo de espera:** un reporte de segundos no debe frenar el registro de un paquete.

Como en Sistemas Operativos

Mismo problema: condiciones de carrera. Pero nadie marca secciones críticas: cada transacción se escribe como si estuviera sola y el **control de concurrencia** restringe el intercalado.

¿Por qué no una transacción después de otra?

- **Throughput:** mientras una espera al disco, otra usa la CPU.
- **Tiempo de espera:** un reporte de segundos no debe frenar el registro de un paquete.

> **Como en Sistemas Operativos**
>
> Mismo problema: condiciones de carrera. Pero nadie marca secciones críticas: cada transacción se escribe como si estuviera sola y el **control de concurrencia** restringe el intercalado.

Una carrera. Envío 501, 50 soles. A cobra 5 de embalaje; B descuenta 10.

lost update

```
T1: read(P)              -- 50
T2:       read(P)        -- 50
T1:             write(P) -- 55
T2:             write(P) -- 40
```

¿Por qué no una transacción después de otra?

- **Throughput:** mientras una espera al disco, otra usa la CPU.
- **Tiempo de espera:** un reporte de segundos no debe frenar el registro de un paquete.

> **Como en Sistemas Operativos**
>
> Mismo problema: condiciones de carrera. Pero nadie marca secciones críticas: cada transacción se escribe como si estuviera sola y el **control de concurrencia** restringe el intercalado.

Una carrera. Envío 501, 50 soles. A cobra 5 de embalaje; B descuenta 10.

lost update

```
T1: read(P)              -- 50
T2:       read(P)        -- 50
T1:             write(P) -- 55
T2:             write(P) -- 40
```

En serie: 45. Aquí: 40. El embalaje se perdió: **lost update**.

Schedules

Schedule

Una secuencia de las operaciones de un conjunto de transacciones que contiene todas las operaciones de cada una y respeta su orden dentro de cada transacción.

`r1(A)`	T1 lee A
`w2(B)`	T2 escribe B
`c1`	T1 hace commit
`a2`	T2 aborta

El lost update: `r1(P) r2(P) w1(P) w2(P)`.

Los cálculos no aparecen. No es un descuido: el DBMS tampoco los ve.

Hasta la sesión B, todas hacen commit: omitimos los `c`.

Schedules seriales y serializables

- **Serial:** una transacción completa, después otra. Con n transacciones hay $n!$.

serial T1, T2	`r1(P) w1(P) r2(P) w2(P)`	$50 + 5 - 10 = 45$
serial T2, T1	`r2(P) w2(P) r1(P) w1(P)`	$50 - 10 + 5 = 45$

Schedules seriales y serializables

- **Serial:** una transacción completa, después otra. Con n transacciones hay $n!$.

serial T1, T2	`r1(P) w1(P) r2(P) w2(P)`	$50 + 5 - 10 = 45$
serial T2, T1	`r2(P) w2(P) r1(P) w1(P)`	$50 - 10 + 5 = 45$

- Con “10 % de descuento”: 49.50 o 50. Los dos son correctos.
- Todo schedule serial preserva la consistencia.

Schedules seriales y serializables

- **Serial:** una transacción completa, después otra. Con n transacciones hay $n!$.

serial T1, T2	`r1(P) w1(P) r2(P) w2(P)`	$50 + 5 - 10 = 45$
serial T2, T1	`r2(P) w2(P) r1(P) w1(P)`	$50 - 10 + 5 = 45$

- Con “10 % de descuento”: 49.50 o 50. Los dos son correctos.
- Todo schedule serial preserva la consistencia.

Serializable

Equivalente a *algún* schedule serial de las mismas transacciones: por conflictos (la que se usa) o por vistas.

Schedules seriales y serializables

- **Serial:** una transacción completa, después otra. Con n transacciones hay $n!$.

serial T1, T2	`r1(P) w1(P) r2(P) w2(P)`	$50 + 5 - 10 = 45$
serial T2, T1	`r2(P) w2(P) r1(P) w1(P)`	$50 - 10 + 5 = 45$

- Con “10 % de descuento”: 49.50 o 50. Los dos son correctos.
- Todo schedule serial preserva la consistencia.

Serializable

Equivalente a *algún* schedule serial de las mismas transacciones: por conflictos (la que se usa) o por vistas.

Error común: “serializable” no es “se ejecutó en serie” ni “en el orden de llegada”.

SECCIÓN 03

Conflict serializability

03



Operaciones en conflicto

I de Ti y J de Tj, consecutivas, sobre el mismo ítem Q:

I	J	¿Importa?
`read(Q)`	`read(Q)`	no
`read(Q)`	`write(Q)`	sí
`write(Q)`	`read(Q)`	sí
`write(Q)`	`write(Q)`	sí

Conflicto

Dos operaciones de transacciones **distintas**, sobre el **mismo ítem**, y al menos una es una **escritura**.

Operaciones en conflicto

I de Ti y J de Tj, consecutivas, sobre el mismo ítem Q:

I	J	¿Importa?
`read(Q)`	`read(Q)`	no
`read(Q)`	`write(Q)`	sí
`write(Q)`	`read(Q)`	sí
`write(Q)`	`write(Q)`	sí

Conflicto

Dos operaciones de transacciones **distintas**, sobre el **mismo ítem**, y al menos una es una **escritura**.

Pares `r-w`, `w-r`, `w-w`. Dos lecturas, nunca. Dos de la misma transacción, tampoco: su orden es fijo.

Intercambiar operaciones vecinas

Conflict equivalent, conflict serializable

S y S' son equivalentes por conflictos si S se transforma en S' intercambiando operaciones consecutivas que *no* están en conflicto. S es conflict serializable si es equivalente por conflictos a un schedule serial.

```
r1(A) w1(A) r2(A) w2(A) r1(B) w1(B) r2(B) w2(B)
```

Intercambiar operaciones vecinas

Conflict equivalent, conflict serializable

S y S' son equivalentes por conflictos si S se transforma en S' intercambiando operaciones consecutivas que *no* están en conflicto. S es conflict serializable si es equivalente por conflictos a un schedule serial.

```
   r1(A) w1(A) r2(A) w2(A) r1(B) w1(B) r2(B) w2(B)
→  r1(A) w1(A) r2(A) r1(B) w2(A) w1(B) r2(B) w2(B)
```

Intercambiar operaciones vecinas

Conflict equivalent, conflict serializable

S y S' son equivalentes por conflictos si S se transforma en S' intercambiando operaciones consecutivas que *no* están en conflicto. S es conflict serializable si es equivalente por conflictos a un schedule serial.

```
   r1(A) w1(A) r2(A) w2(A) r1(B) w1(B) r2(B) w2(B)
→  r1(A) w1(A) r2(A) r1(B) w2(A) w1(B) r2(B) w2(B)
→  r1(A) w1(A) r1(B) r2(A) w2(A) w1(B) r2(B) w2(B)
```

Intercambiar operaciones vecinas

Conflict equivalent, conflict serializable

S y S' son equivalentes por conflictos si S se transforma en S' intercambiando operaciones consecutivas que *no* están en conflicto. S es conflict serializable si es equivalente por conflictos a un schedule serial.

```
   r1(A) w1(A) r2(A) w2(A) r1(B) w1(B) r2(B) w2(B)
→  r1(A) w1(A) r2(A) r1(B) w2(A) w1(B) r2(B) w2(B)
→  r1(A) w1(A) r1(B) r2(A) w2(A) w1(B) r2(B) w2(B)
→  ...
→  r1(A) w1(A) r1(B) w1(B) r2(A) w2(A) r2(B) w2(B)
```

serial T1, T2

Intercambiar operaciones vecinas

Conflict equivalent, conflict serializable

S y S' son equivalentes por conflictos si S se transforma en S' intercambiando operaciones consecutivas que *no* están en conflicto. S es conflict serializable si es equivalente por conflictos a un schedule serial.

```
   r1(A) w1(A) r2(A) w2(A) r1(B) w1(B) r2(B) w2(B)
→  r1(A) w1(A) r2(A) r1(B) w2(A) w1(B) r2(B) w2(B)
→  r1(A) w1(A) r1(B) r2(A) w2(A) w1(B) r2(B) w2(B)
→  ...
→  r1(A) w1(A) r1(B) w1(B) r2(A) w2(A) r2(B) w2(B)
```

Funciona, pero buscar serial T1, T2

los intercambios es tedioso, y cuando no existen no hay cómo probarlo intentando.

El grafo de precedencia

Grafo de precedencia ***(precedence graph)***

Un nodo por transacción; una arista Ti $\rightarrow$ Tj si una operación de Ti va antes que una de Tj con la que está en conflicto:

- Ti `write(Q)` antes de Tj `read(Q)`;
- Ti `read(Q)` antes de Tj `write(Q)`;
- Ti `write(Q)` antes de Tj `write(Q)`.

El grafo de precedencia

Grafo de precedencia *(precedence graph)*

Un nodo por transacción; una arista Ti $\rightarrow$ Tj si una operación de Ti va antes que una de Tj con la que está en conflicto:

- Ti `write(Q)` antes de Tj `read(Q)`;
- Ti `read(Q)` antes de Tj `write(Q)`;
- Ti `write(Q)` antes de Tj `write(Q)`.

Teorema

S es conflict serializable $\iff$ su grafo no tiene ciclos.
Cualquier **orden topológico** da un schedule serial equivalente.

El grafo de precedencia

Grafo de precedencia *(precedence graph)*

Un nodo por transacción; una arista Ti $\rightarrow$ Tj si una operación de Ti va antes que una de Tj con la que está en conflicto:

- Ti `write(Q)` antes de Tj `read(Q)`;
- Ti `read(Q)` antes de Tj `write(Q)`;
- Ti `write(Q)` antes de Tj `write(Q)`.

Teorema

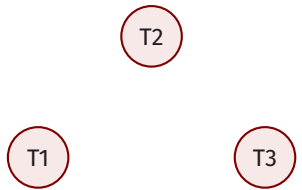
S es conflict serializable $\iff$ su grafo no tiene ciclos. Cualquier **orden topológico** da un schedule serial equivalente.

- Ciclo T1 $\rightarrow$ T2 $\rightarrow$ T1: T1 antes que T2 y T2 antes que T1. Imposible.
- Sin ciclo: la transacción sin aristas de entrada se lleva al principio con intercambios; se repite.

Ejemplo resuelto

A = fila del envío 501, B = fila del viaje 12.

```
S = r1(A) r2(B) w1(A) r3(A) w2(B) r3(B) w3(A) r1(B)
```

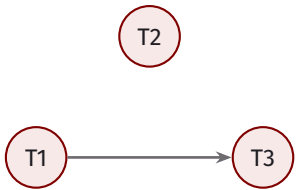


Ejemplo resuelto

A = fila del envío 501, B = fila del viaje 12.

```
S = r1(A) r2(B) w1(A) r3(A) w2(B) r3(B) w3(A) r1(B)
```

- Sobre A: `r1(A) w1(A) r3(A) w3(A)`.
  `r1(A)`–`w3(A)`, `w1(A)`–`r3(A)`, `w1(A)`–`w3(A)`: **T1** $\rightarrow$ **T3**.

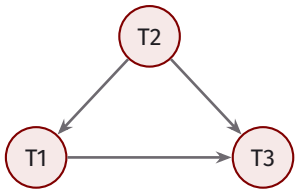


Ejemplo resuelto

A = fila del envío 501, B = fila del viaje 12.

```
S = r1(A) r2(B) w1(A) r3(A) w2(B) r3(B) w3(A) r1(B)
```

- Sobre A: `r1(A) w1(A) r3(A) w3(A)`.
  `r1(A)–w3(A)`, `w1(A)–r3(A)`, `w1(A)–w3(A)`: **T1** $\rightarrow$ **T3**.
- Sobre B: `r2(B) w2(B) r3(B) r1(B)`.
  `w2(B)–r3(B)`: **T2** $\rightarrow$ **T3**; `w2(B)–r1(B)`: **T2** $\rightarrow$ **T1**.

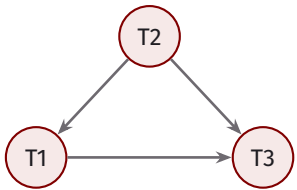


Ejemplo resuelto

A = fila del envío 501, B = fila del viaje 12.

```
S = r1(A) r2(B) w1(A) r3(A) w2(B) r3(B) w3(A) r1(B)
```

- Sobre A: `r1(A) w1(A) r3(A) w3(A)`. `r1(A)`–`w3(A)`, `w1(A)`–`r3(A)`, `w1(A)`–`w3(A)`: **T1** $\rightarrow$ **T3**.
- Sobre B: `r2(B) w2(B) r3(B) r1(B)`. `w2(B)`–`r3(B)`: **T2** $\rightarrow$ **T3**; `w2(B)`–`r1(B)`: **T2** $\rightarrow$ **T1**.
- `r3(B)` y `r1(B)`: dos lecturas, sin conflicto.

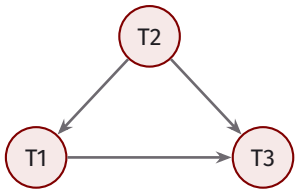


Ejemplo resuelto

A = fila del envío 501, B = fila del viaje 12.

```
S = r1(A) r2(B) w1(A) r3(A) w2(B) r3(B) w3(A) r1(B)
```

- Sobre A: `r1(A) w1(A) r3(A) w3(A)`. `r1(A)`–`w3(A)`, `w1(A)`–`r3(A)`, `w1(A)`–`w3(A)`: **T1** → **T3**.
- Sobre B: `r2(B) w2(B) r3(B) r1(B)`. `w2(B)`–`r3(B)`: **T2** → **T3**; `w2(B)`–`r1(B)`: **T2** → **T1**.
- `r3(B)` y `r1(B)`: dos lecturas, sin conflicto.



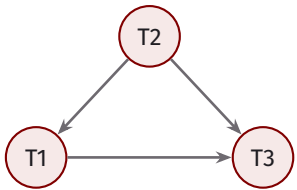
Sin ciclo. Único orden topológico: T2, T1, T3.

Ejemplo resuelto

A = fila del envío 501, B = fila del viaje 12.

```
S = r1(A) r2(B) w1(A) r3(A) w2(B) r3(B) w3(A) r1(B)
```

- Sobre A: `r1(A) w1(A) r3(A) w3(A)`. `r1(A)`–`w3(A)`, `w1(A)`–`r3(A)`, `w1(A)`–`w3(A)`: **T1** → **T3**.
- Sobre B: `r2(B) w2(B) r3(B) r1(B)`. `w2(B)`–`r3(B)`: **T2** → **T3**; `w2(B)`–`r1(B)`: **T2** → **T1**.
- `r3(B)` y `r1(B)`: dos lecturas, sin conflicto.



Sin ciclo. Único orden topológico: T2, T1, T3.

Comprobación: `r1(B)` lee el B de T2 y `r3(A)` el A de T1, en S y en el serial. T1 empezó primero y va segunda.

graph.mjs

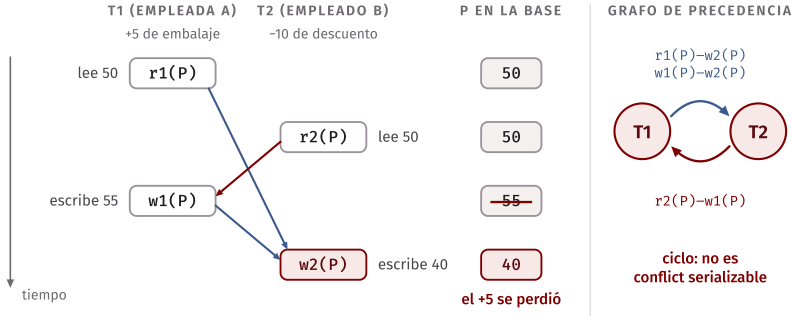
code/week-13

```
$ node graph.mjs "r1(A) r2(B) w1(A) r3(A) w2(B) r3(B) w3(A) r1(B)"
...
aristas:
  T1 → T3  (r1(A)-w3(A), w1(A)-r3(A), w1(A)-w3(A))
  T2 → T1  (w2(B)-r1(B))
  T2 → T3  (w2(B)-r3(B))

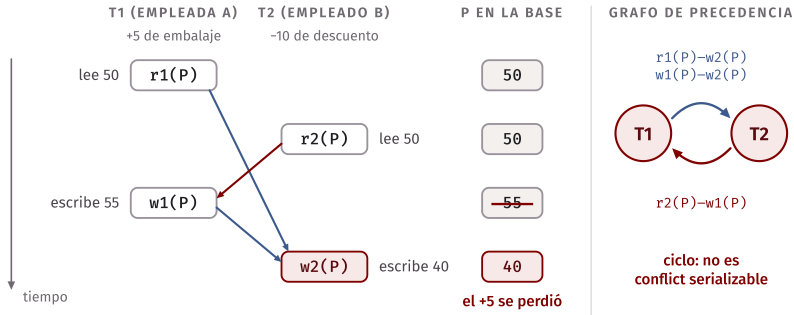
conflict serializable; orden serial equivalente: T2, T1, T3
  r2(B) w2(B) r1(A) w1(A) r1(B) r3(A) r3(B) w3(A)
$ node graph.mjs "r1(A) r2(A) w1(A) w2(A)"
...
aristas:
  T1 → T2  (r1(A)-w2(A), w1(A)-w2(A))
  T2 → T1  (r2(A)-w1(A))

no es conflict serializable; ciclo: T1 → T2 → T1
```

El lost update tiene un ciclo



El lost update tiene un ciclo



T2 leyó P *antes* de que T1 lo escribiera (va antes) y escribió *después* (va después). No hay orden serial que respete las dos: la escritura de T1 se pierde.

El grafo no mira los valores

A y B: precios de 501 y 502. T1 pasa 5 soles de A a B; T2 pasa 2 de B a A.

```
r1(A) w1(A) r2(B) w2(B) r1(B) w1(B) r2(A) w2(A)
```

El grafo no mira los valores

A y B: precios de 501 y 502. T1 pasa 5 soles de A a B; T2 pasa 2 de B a A.

```
r1(A) w1(A) r2(B) w2(B) r1(B) w1(B) r2(A) w2(A)
```

- Con A = 50, B = 45: final A = 47, B = 48. Igual que los dos órdenes seriales: sumas y restas conmutan.

El grafo no mira los valores

A y B: precios de 501 y 502. T1 pasa 5 soles de A a B; T2 pasa 2 de B a A.

```
r1(A) w1(A) r2(B) w2(B) r1(B) w1(B) r2(A) w2(A)
```

- Con A = 50, B = 45: final A = 47, B = 48. Igual que los dos órdenes seriales: sumas y restas conmutan.
- Pero A da T1 $\rightarrow$ T2 y B da T2 $\rightarrow$ T1: ciclo. No es conflict serializable.

El grafo no mira los valores

A y B: precios de 501 y 502. T1 pasa 5 soles de A a B; T2 pasa 2 de B a A.

```
r1(A) w1(A) r2(B) w2(B) r1(B) w1(B) r2(A) w2(A)
```

- Con A = 50, B = 45: final A = 47, B = 48. Igual que los dos órdenes seriales: sumas y restas conmutan.
- Pero A da T1 $\rightarrow$ T2 y B da T2 $\rightarrow$ T1: ciclo. No es conflict serializable.
- Conflict serializability es condición **suficiente**, no necesaria. El DBMS solo ve lecturas y escrituras: lo rechazaría igual.

El grafo no mira los valores

A y B: precios de 501 y 502. T1 pasa 5 soles de A a B; T2 pasa 2 de B a A.

```
r1(A) w1(A) r2(B) w2(B) r1(B) w1(B) r2(A) w2(A)
```

- Con A = 50, B = 45: final A = 47, B = 48. Igual que los dos órdenes seriales: sumas y restas conmutan.
- Pero A da T1 $\rightarrow$ T2 y B da T2 $\rightarrow$ T1: ciclo. No es conflict serializable.
- Conflict serializability es condición **suficiente**, no necesaria. El DBMS solo ve lecturas y escrituras: lo rechazaría igual.

Error común

"Probar con números" no decide nada. Un schedule conflict serializable da un resultado serial con *cualquier* valor inicial y cualquier cálculo.

¿Por qué conflictos?

1. Solo depende del orden de lecturas y escrituras: lo que el DBMS ve.
2. Se decide en tiempo polinomial con el grafo (búsqueda de ciclos).
3. Se puede **garantizar por construcción**: two-phase locking (semana 14) solo produce schedules con grafo acíclico.

En PostgreSQL

Ningún DBMS construye el grafo mientras ejecuta. Impide de antemano los patrones que formarían un ciclo (locks), o los detecta de forma aproximada y aborta a alguien (`SERIALIZABLE`). El grafo es para razonar.

View serializability, en breve

```
S'' = r1(A) w2(A) w1(A) w3(A)
```

- Ciclo T1 $\leftrightarrow$ T2: no es conflict serializable.

View serializability, en breve

```
S'' = r1(A) w2(A) w1(A) w3(A)
```

- Ciclo T1 $\leftrightarrow$ T2: no es conflict serializable.
- Pero T1 lee el valor inicial y T3 hace la última escritura, igual que en el serial T1, T2, T3. Lo de T2 no lo lee nadie.
- `w2(A)` es una **escritura ciega** *(blind write)*.

View serializability, en breve

```
S'' = r1(A) w2(A) w1(A) w3(A)
```

- Ciclo T1 $\leftrightarrow$ T2: no es conflict serializable.
- Pero T1 lee el valor inicial y T3 hace la última escritura, igual que en el serial T1, T2, T3. Lo de T2 no lo lee nadie.
- `w2(A)` es una **escritura ciega** *(blind write)*.

View equivalent

Para cada ítem Q: (1) las mismas lecturas del valor inicial; (2) cada lectura lee de la misma escritura; (3) la misma última escritura.

View serializability, en breve

```
S'' = r1(A) w2(A) w1(A) w3(A)
```

- Ciclo T1 $\leftrightarrow$ T2: no es conflict serializable.
- Pero T1 lee el valor inicial y T3 hace la última escritura, igual que en el serial T1, T2, T3. Lo de T2 no lo lee nadie.
- `w2(A)` es una **escritura ciega** *(blind write)*.

View equivalent

Para cada ítem Q: (1) las mismas lecturas del valor inicial; (2) cada lectura lee de la misma escritura; (3) la misma última escritura.

- Conflict serializable $\Rightarrow$ view serializable; no al revés.
- Decidirla es NP-completo. Los sistemas usan conflictos.

SECCIÓN 04

Recuperabilidad

04

Schedules recuperables

T1 cambia el precio del 501. T2 lo lee, emite la boleta y hace commit. Después T1 aborta:

```
w1(A) r2(A) c2 a1
```

Schedules recuperables

T1 cambia el precio del 501. T2 lo lee, emite la boleta y hace commit. Después T1 aborta:

```
w1(A) r2(A) c2 a1
```

- T2 leyó un precio que nunca existió, y su commit no se deshace.
- Y el schedule es serializable: solo T1 $\rightarrow$ T2.

Schedules recuperables

T1 cambia el precio del 501. T2 lo lee, emite la boleta y hace commit. Después T1 aborta:

```
w1(A) r2(A) c2 a1
```

- T2 leyó un precio que nunca existió, y su commit no se deshace.
- Y el schedule es serializable: solo T1 → T2.

Recuperable ***(recoverable)***

Tj **lee de** Ti si lee un ítem cuya última escritura previa es de Ti. Recuperable: si Tj lee de Ti, el commit de Ti va antes que el commit de Tj.

Schedules recuperables

T1 cambia el precio del 501. T2 lo lee, emite la boleta y hace commit. Después T1 aborta:

```
w1(A) r2(A) c2 a1
```

- T2 leyó un precio que nunca existió, y su commit no se deshace.
- Y el schedule es serializable: solo T1 → T2.

Recuperable ***(recoverable)***

Tj **lee de** Ti si lee un ítem cuya última escritura previa es de Ti. Recuperable: si Tj lee de Ti, el commit de Ti va antes que el commit de Tj.

`w1(A) r2(A) c1 c2` sí es recuperable. El DBMS debe producir solo schedules recuperables.

Rollback en cascada y cascadeless

```
w1(A) r2(A) w2(B) r3(B) a1
```

Recuperable (nadie hizo commit), pero abortar T1 obliga a abortar T2, que leyó de T1, y T3, que leyó de T2: **rollback en cascada**.

Rollback en cascada y cascadeless

```
w1(A) r2(A) w2(B) r3(B) a1
```

Recuperable (nadie hizo commit), pero abortar T1 obliga a abortar T2, que leyó de T1, y T3, que leyó de T2: **rollback en cascada**.

Cascadeless

Si Tj lee de Ti, el commit de Ti va antes de la lectura de Tj: solo se lee lo confirmado. Todo cascadeless es recuperable.

Rollback en cascada y cascadeless

`w1(A) r2(A) w2(B) r3(B) a1`

Recuperable (nadie hizo commit), pero abortar T1 obliga a abortar T2, que leyó de T1, y T3, que leyó de T2: **rollback en cascada**.

Cascadeless

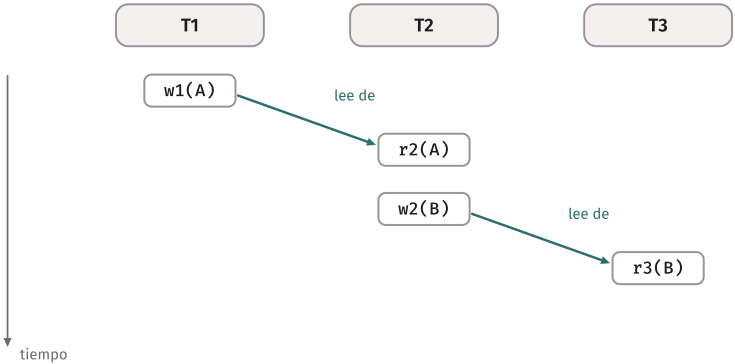
Si Tj lee de Ti, el commit de Ti va antes de la lectura de Tj: solo se lee lo confirmado. Todo cascadeless es recuperable.

code/week-13

```
$ node graph.mjs "w1(A) r2(A) c2 a1"
...
recuperable: no; T2 lee A de T1
  (w1(A) ... r2(A)) y hace commit
  antes que T1
cascadeless: no; T2 lee A de T1
  (w1(A) ... r2(A)) antes del
  commit de T1
```

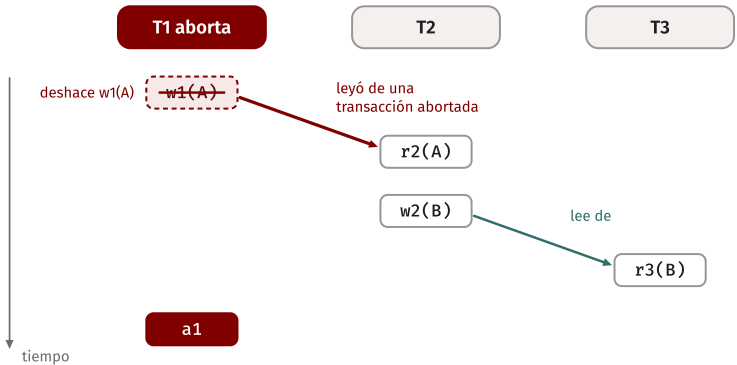
(líneas partidas a mano para que entren)

Un abort que arrastra a los demás



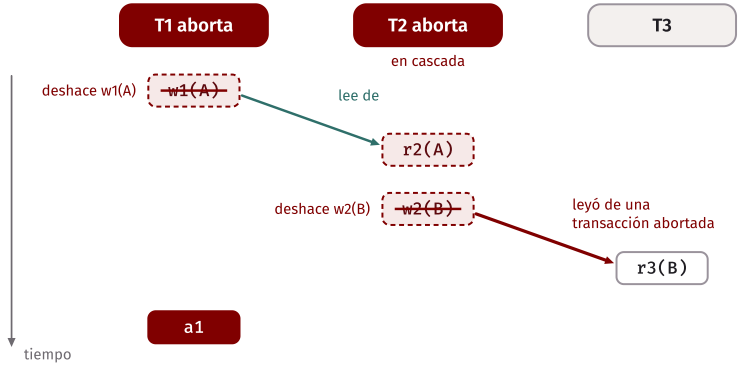
T2 lee A de T1 y T3 lee B de T2; nadie hizo commit todavía

Un abort que arrastra a los demás



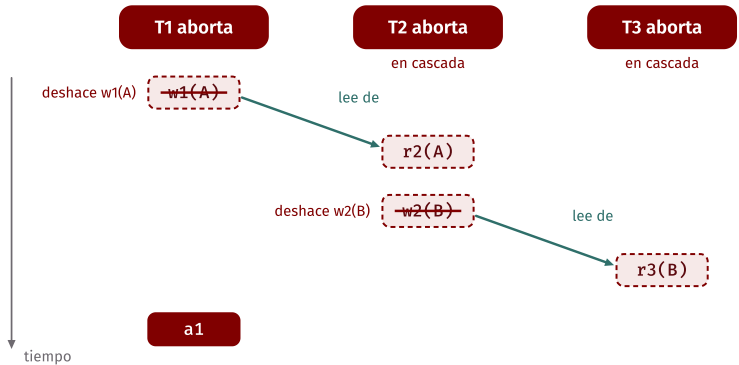
a1: T1 aborta, y hay que deshacer su escritura de A

Un abort que arrastra a los demás



T2 leyó un A que nunca existió: también aborta, y se deshace w2(B)

Un abort que arrastra a los demás



T3 leyó un B que nunca existió: también aborta

recuperable (nadie hizo commit), pero no cascadeless

Dos ejes distintos

`w1(A) c1 r2(A) w2(B) c2 r3(B) c3`	cascadeless (y recuperable)
`w1(A) r2(A) c1 c2`	recuperable, no cascadeless
`w1(A) r2(A) c2 a1`	ni recuperable ni cascadeless

Idea clave

Serializabilidad mira el orden de lecturas y escrituras; recuperabilidad, dónde caen los commits. El control de concurrencia tiene que dar las dos (strict 2PL, semana 14). “Solo leer lo confirmado” es no tener **dirty reads**.

SECCIÓN 05

Niveles de aislamiento

05

Las anomalías del estándar SQL

Serializable cuesta esperas o abortos. SQL permite elegir un nivel más débil por transacción. Ningún nivel permite dirty writes (sec. 17.8).

Las anomalías del estándar SQL

Serializable cuesta esperas o abortos. SQL permite elegir un nivel más débil por transacción. Ningún nivel permite dirty writes (sec. 17.8).

- **Dirty read:** T1 lee lo que T2 escribió sin commit.

Las anomalías del estándar SQL

Serializable cuesta esperas o abortos. SQL permite elegir un nivel más débil por transacción. Ningún nivel permite dirty writes (sec. 17.8).

- **Dirty read:** T1 lee lo que T2 escribió sin commit.
- **Non-repeatable read:** T1 lee una fila, T2 la cambia y confirma, T1 la relee y ve otro valor.

Las anomalías del estándar SQL

Serializable cuesta esperas o abortos. SQL permite elegir un nivel más débil por transacción. Ningún nivel permite dirty writes (sec. 17.8).

- **Dirty read:** T1 lee lo que T2 escribió sin commit.
- **Non-repeatable read:** T1 lee una fila, T2 la cambia y confirma, T1 la relee y ve otro valor.
- **Phantom:** T1 lee las filas que cumplen una condición, T2 inserta una que la cumple, T1 repite y aparece una fila nueva.

Las anomalías del estándar SQL

Serializable cuesta esperas o abortos. SQL permite elegir un nivel más débil por transacción. Ningún nivel permite dirty writes (sec. 17.8).

- **Dirty read:** T1 lee lo que T2 escribió sin commit.
- **Non-repeatable read:** T1 lee una fila, T2 la cambia y confirma, T1 la relee y ve otro valor.
- **Phantom:** T1 lee las filas que cumplen una condición, T2 inserta una que la cumple, T1 repite y aparece una fila nueva.

Filas o predicados

Contra un non-repeatable read basta proteger las filas leídas. Contra un phantom no: la fila de T2 no existía. El conflicto es sobre el predicado `trip_id = 5708` (sec. 17.10; semana 14).

Los cuatro niveles

Nivel	Dirty read	Non-repeatable	Phantom
`SERIALIZABLE`	no	no	no
`REPEATABLE READ`	no	no	posible
`READ COMMITTED`	no	posible	posible
`READ UNCOMMITTED`	posible	posible	posible

Los cuatro niveles

Nivel	Dirty read	Non-repeatable	Phantom
`SERIALIZABLE`	no	no	no
`REPEATABLE READ`	no	no	posible
`READ COMMITTED`	no	posible	posible
`READ UNCOMMITTED`	posible	posible	posible

- “Posible” = el estándar lo permite, no lo exige. Cada motor elige.
- `SERIALIZABLE` exige ejecución serializable; evitar las tres no alcanza.

Los cuatro niveles

Nivel	Dirty read	Non-repeatable	Phantom
`SERIALIZABLE`	no	no	no
`REPEATABLE READ`	no	no	posible
`READ COMMITTED`	no	posible	posible
`READ UNCOMMITTED`	posible	posible	posible

- "Posible" = el estándar lo permite, no lo exige. Cada motor elige.
- `SERIALIZABLE` exige ejecución serializable; evitar las tres no alcanza.

Dos que no están en la lista (Berenson et al., 1995):

- **Lost update:** `r1(A) r2(A) w1(A) w2(A)`.
- **Write skew:** leen lo mismo, verifican una regla, escriben *filas distintas*. Ciclo con dos aristas `r-w`.

SECCIÓN 06

Lo que hace PostgreSQL

06



Los niveles en PostgreSQL

Nivel	Dirty	Non-rep.	Phantom	Serialization anomaly
`READ UNCOMMITTED`	no en PG	posible	posible	posible
`READ COMMITTED`	no	posible	posible	posible
`REPEATABLE READ`	no	no	no en PG	posible
`SERIALIZABLE`	no	no	no	no

Los niveles en PostgreSQL

Nivel	Dirty	Non-rep.	Phantom	Serialization anomaly
`READ UNCOMMITTED`	no en PG	posible	posible	posible
`READ COMMITTED`	no	posible	posible	posible
`REPEATABLE READ`	no	no	no en PG	posible
`SERIALIZABLE`	no	no	no	no

- `READ UNCOMMITTED` se comporta como `READ COMMITTED`.
- `READ COMMITTED` (por defecto): un **snapshot por sentencia**.
- `REPEATABLE READ`: **un snapshot para toda la transacción** = snapshot isolation.
- `SERIALIZABLE`: snapshot isolation + detección de ciclos posibles (SSI).

Los niveles en PostgreSQL

Nivel	Dirty	Non-rep.	Phantom	Serialization anomaly
`READ UNCOMMITTED`	no en PG	posible	posible	posible
`READ COMMITTED`	no	posible	posible	posible
`REPEATABLE READ`	no	no	no en PG	posible
`SERIALIZABLE`	no	no	no	no

- `READ UNCOMMITTED` se comporta como `READ COMMITTED`.
- `READ COMMITTED` (por defecto): un **snapshot por sentencia**.
- `REPEATABLE READ`: **un snapshot para toda la transacción** = snapshot isolation.
- `SERIALIZABLE`: snapshot isolation + detección de ciclos posibles (SSI).

Cómo leer lo que sigue

Dos `psql`, A y B, manejados por `psql2.mjs`; el script solo agrega `-- B espera` y `-- B continúa`. Partida: 501 = 50.00, 502 = 45.00.

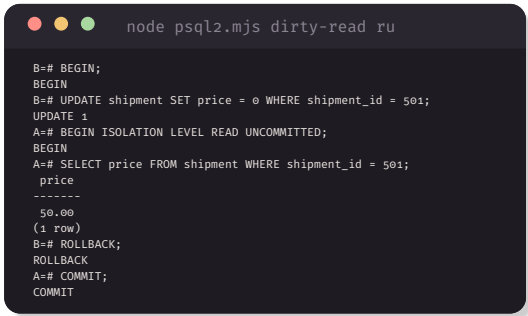
Sin dirty reads

```
node psql2.mjs dirty-read ru
```

```
B=# BEGIN;
BEGIN
B=# UPDATE shipment SET price = 0 WHERE shipment_id = 501;
UPDATE 1
A=# BEGIN ISOLATION LEVEL READ UNCOMMITTED;
BEGIN
A=# SELECT price FROM shipment WHERE shipment_id = 501;
 price
-------
 50.00
(1 row)
B=# ROLLBACK;
ROLLBACK
A=# COMMIT;
COMMIT
```

- A ve 50.00, el último valor confirmado: `READ UNCOMMITTED` se comporta como `READ COMMITTED`.

Sin dirty reads



- A ve 50.00, el último valor confirmado: `READ UNCOMMITTED` se comporta como `READ COMMITTED`.
- Y A no esperó, aunque B tiene la fila bloqueada para escribir: lee la versión anterior.

Sin dirty reads

```
node psql2.mjs dirty-read ru

B=# BEGIN;
BEGIN
B=# UPDATE shipment SET price = 0 WHERE shipment_id = 501;
UPDATE 1
A=# BEGIN ISOLATION LEVEL READ UNCOMMITTED;
BEGIN
A=# SELECT price FROM shipment WHERE shipment_id = 501;
 price
-------
 50.00
(1 row)
B=# ROLLBACK;
ROLLBACK
A=# COMMIT;
COMMIT
```

- A ve 50.00, el último valor confirmado: `READ UNCOMMITTED` se comporta como `READ COMMITTED`.
- Y A no esperó, aunque B tiene la fila bloqueada para escribir: lee la versión anterior.
- Con locks y sin versiones, A esperaría el fin de B. Primer adelanto de MVCC.

`READ COMMITTED`: una foto por sentencia

node psql2.mjs non-repeatable-read rc

```
A=# BEGIN ISOLATION LEVEL READ COMMITTED;
BEGIN
A=# SELECT price FROM shipment WHERE shipment_id = 501;
 price
-------
 50.00
(1 row)
B=# UPDATE shipment SET price = price + 10 WHERE shipment_id = 501;
UPDATE 1
A=# SELECT price FROM shipment WHERE shipment_id = 501;
 price
-------
 60.00
(1 row)
A=# COMMIT;
COMMIT
```

Non-repeatable read.

node psql2.mjs phantom rc

```
A=# BEGIN ISOLATION LEVEL READ COMMITTED;
BEGIN
A=# SELECT count(*), sum(weight_kg) FROM shipment
      WHERE trip_id = 5708;
 count |  sum
-------+--------
     6 | 166.60
(1 row)
B=# INSERT INTO shipment (...) VALUES (12001, ..., 5708, ...);
INSERT 0 1
A=# SELECT count(*), sum(weight_kg) FROM shipment
      WHERE trip_id = 5708;
 count |  sum
-------+--------
     7 | 176.60
(1 row)
```

Phantom. (sentencias abreviadas para que entren)

Cada sentencia ve un estado consistente. El problema: decidir con lo leído en una sentencia y actuar en otra.

REPEATABLE READ: una foto para toda la transacción

node psql2.mjs non-repeatable-read rr

```
A=# BEGIN ISOLATION LEVEL REPEATABLE READ;
BEGIN
A=# SELECT price FROM shipment WHERE shipment_id = 501;
 price
-------
 50.00
(1 row)
B=# UPDATE shipment SET price = price + 10 WHERE shipment_id = 501;
UPDATE 1
A=# SELECT price FROM shipment WHERE shipment_id = 501;
 price
-------
 50.00
(1 row)
A=# COMMIT;
COMMIT
```

node psql2.mjs phantom rr

```
...
A=# SELECT count(*), sum(weight_kg) FROM shipment
      WHERE trip_id = 5708;
 count |  sum
-------+--------
     6 | 166.60
(1 row)
B=# INSERT INTO shipment (...) VALUES (12001, ..., 5708, ...);
INSERT 0 1
A=# SELECT count(*), sum(weight_kg) FROM shipment
      WHERE trip_id = 5708;
 count |  sum
-------+--------
     6 | 166.60
(1 row)
```

Ni non-repeatable read ni phantom, aunque el estándar lo permitiría: se lee todo de un snapshot fijo. Nadie esperó.

¿Cuándo se toma el snapshot?

node psql2.mjs snapshot-start rr

```
A=# BEGIN ISOLATION LEVEL REPEATABLE READ;
BEGIN
B=# UPDATE shipment SET price = price + 10 WHERE shipment_id = 501;
UPDATE 1
A=# SELECT price FROM shipment WHERE shipment_id = 501;
 price
-------
 60.00
(1 row)
B=# UPDATE shipment SET price = price + 10 WHERE shipment_id = 501;
UPDATE 1
A=# SELECT price FROM shipment WHERE shipment_id = 501;
 price
-------
 60.00
(1 row)
A=# COMMIT;
COMMIT
```

En PostgreSQL

El snapshot de `REPEATABLE READ` se toma en la **primera consulta**, no en el `BEGIN`.

A ve el primer `UPDATE` de B y no el segundo. Para el orden serial da igual: A queda entre los dos.

Lost update en `READ COMMITTED`

node psql2.mjs lost-update rc

```
A=# BEGIN ISOLATION LEVEL READ COMMITTED;
BEGIN
B=# BEGIN ISOLATION LEVEL READ COMMITTED;
BEGIN
A=# SELECT price FROM shipment WHERE shipment_id = 501;
 price
-------
 50.00
(1 row)
B=# SELECT price FROM shipment WHERE shipment_id = 501;
 price
-------
 50.00
(1 row)
A=# UPDATE shipment SET price = 55 WHERE shipment_id = 501;
UPDATE 1
```

(continuación)

```
B=# UPDATE shipment SET price = 40 WHERE shipment_id = 501;
-- B espera
A=# COMMIT;
COMMIT
-- B continúa
UPDATE 1
B=# COMMIT;
COMMIT
A=# SELECT price FROM shipment WHERE shipment_id = 501;
 price
-------
 40.00
(1 row)
```

Lost update en `READ COMMITTED`

node psql2.mjs lost-update rc

```
A=# BEGIN ISOLATION LEVEL READ COMMITTED;
BEGIN
B=# BEGIN ISOLATION LEVEL READ COMMITTED;
BEGIN
A=# SELECT price FROM shipment WHERE shipment_id = 501;
 price
-------
 50.00
(1 row)
B=# SELECT price FROM shipment WHERE shipment_id = 501;
 price
-------
 50.00
(1 row)
A=# UPDATE shipment SET price = 55 WHERE shipment_id = 501;
UPDATE 1
```

(continuación)

```
B=# UPDATE shipment SET price = 40 WHERE shipment_id = 501;
-- B espera
A=# COMMIT;
COMMIT
-- B continúa
UPDATE 1
B=# COMMIT;
COMMIT
A=# SELECT price FROM shipment WHERE shipment_id = 501;
 price
-------
 40.00
(1 row)
```

`r1(A) r2(A) w1(A) c1 w2(A) c2`: el ciclo de antes. B esperó el lock de la fila (no hay dirty write), pero ya había leído 50 y escribe un valor fijo.

Lost update en `REPEATABLE READ` y `SERIALIZABLE`

```
node psql2.mjs lost-update rr

...
A=# UPDATE shipment SET price = 55 WHERE shipment_id = 501;
UPDATE 1
B=# UPDATE shipment SET price = 40 WHERE shipment_id = 501;
-- B espera
A=# COMMIT;
COMMIT
-- B continúa
ERROR:  could not serialize access due to concurrent update
B=# COMMIT;
ROLLBACK
```

```
node psql2.mjs lost-update rr --verbose

-- B continúa
ERROR:  40001: could not serialize access due to concurrent update
LOCATION:  ExecUpdate, nodeModifyTable.c:2602
```

- La fila cambió después del snapshot de B: aborta.
- Snapshot isolation: de dos que escriben la misma fila, una sola confirma.
- `COMMIT` de una abortada: `ROLLBACK`.
- `40001` = `serialization_failure`.
- El precio queda en 55.00.
- En `SERIALIZABLE`: mismo error, misma sentencia (el `UPDATE`).

Los niveles fuertes no bloquean más: abortan

Idea clave

En PostgreSQL leer nunca espera, en ningún nivel. Al subir de nivel no hay más esperas: hay más transacciones que terminan en `40001`. Una aplicación en `REPEATABLE READ` o `SERIALIZABLE` tiene que estar preparada para reintentar.

Los niveles fuertes no bloquean más: abortan

Idea clave

En PostgreSQL leer nunca espera, en ningún nivel. Al subir de nivel no hay más esperas: hay más transacciones que terminan en `40001`. Una aplicación en `REPEATABLE READ` o `SERIALIZABLE` tiene que estar preparada para reintentar.

Para el lost update hay dos arreglos que no abortan a nadie, en `READ COMMITTED`:

1. una sola sentencia: `UPDATE ... SET price = price + 5`;
2. leer con lock: `SELECT ... FOR UPDATE`.

Arreglo 1: una sola sentencia

```
node psql2.mjs atomic-update rc

A=# BEGIN ISOLATION LEVEL READ COMMITTED;
BEGIN
B=# BEGIN ISOLATION LEVEL READ COMMITTED;
BEGIN
A=# UPDATE shipment SET price = price + 5 WHERE shipment_id = 501;
UPDATE 1
B=# UPDATE shipment SET price = price - 10 WHERE shipment_id = 501;
-- B espera
A=# COMMIT;
COMMIT
-- B continúa
UPDATE 1
B=# COMMIT;
COMMIT
A=# SELECT price FROM shipment WHERE shipment_id = 501;
 price
-------
 45.00
(1 row)
```

- $50 + 5 - 10 = 45$.
- Cuando A confirma, B **relee la versión confirmada** (55.00), revisa el `WHERE` y aplica `- 10`. Regla de `READ COMMITTED` para `UPDATE`, `DELETE` y `FOR UPDATE`.

Arreglo 1: una sola sentencia

node psql2.mjs atomic-update rc

```
A=# BEGIN ISOLATION LEVEL READ COMMITTED;
BEGIN
B=# BEGIN ISOLATION LEVEL READ COMMITTED;
BEGIN
A=# UPDATE shipment SET price = price + 5 WHERE shipment_id = 501;
UPDATE 1
B=# UPDATE shipment SET price = price - 10 WHERE shipment_id = 501;
-- B espera
A=# COMMIT;
COMMIT
-- B continúa
UPDATE 1
B=# COMMIT;
COMMIT
A=# SELECT price FROM shipment WHERE shipment_id = 501;
 price
-------
 45.00
(1 row)
```

- $50 + 5 - 10 = 45$.
- Cuando A confirma, B **relee la versión confirmada** (55.00), revisa el `WHERE` y aplica `- 10`. Regla de `READ COMMITTED` para `UPDATE`, `DELETE` y `FOR UPDATE`.

Error común

"`READ COMMITTED` siempre pierde actualizaciones." No: las pierde el read-then-write de la aplicación. Y en `REPEATABLE READ` este mismo `UPDATE` *falla* con `40001`.

Arreglo 2: `SELECT ... FOR UPDATE`

```
node psql2.mjs for-update rc

A=# SELECT price FROM shipment WHERE shipment_id = 501 FOR UPDATE;
 price
-------
 50.00
(1 row)
B=# SELECT price FROM shipment WHERE shipment_id = 501 FOR UPDATE;
-- B espera
A=# UPDATE shipment SET price = 55 WHERE shipment_id = 501;
UPDATE 1
A=# COMMIT;
COMMIT
-- B continúa
 price
-------
 55.00
(1 row)
B=# UPDATE shipment SET price = 45 WHERE shipment_id = 501;
UPDATE 1
```

- B espera en la lectura y lee 55.00.
- Queda `r1(A) w1(A) c1 r2(A) w2(A) c2`: serial.

Arreglo 2: `SELECT ... FOR UPDATE`

```
node psql2.mjs for-update rc

A=# SELECT price FROM shipment WHERE shipment_id = 501 FOR UPDATE;
 price
-------
 50.00
(1 row)
B=# SELECT price FROM shipment WHERE shipment_id = 501 FOR UPDATE;
-- B espera
A=# UPDATE shipment SET price = 55 WHERE shipment_id = 501;
UPDATE 1
A=# COMMIT;
COMMIT
-- B continúa
 price
-------
 55.00
(1 row)
B=# UPDATE shipment SET price = 45 WHERE shipment_id = 501;
UPDATE 1
```

- B espera en la lectura y lee 55.00.
- Queda `r1(A) w1(A) c1 r2(A) w2(A) c2`: serial.

Arreglo	¿Bloquea?	¿Aborta?
`FOR UPDATE` (RC)	sí, la lectura	no
RR o `SERIALIZABLE`	sí, el `UPDATE`	sí, `40001`
`price = price + 5` (RC)	sí, el `UPDATE`	no

Write skew en REPEATABLE READ

```
node psql2.mjs write-skew rr

A=# SELECT sum(price) FROM shipment WHERE shipment_id IN (501, 502);
  sum
-------
 95.00
(1 row)
B=# SELECT sum(price) FROM shipment WHERE shipment_id IN (501, 502);
  sum
-------
 95.00
(1 row)
A=# UPDATE shipment SET price = price - 5 WHERE shipment_id = 501;
UPDATE 1
B=# UPDATE shipment SET price = price - 5 WHERE shipment_id = 502;
UPDATE 1
A=# COMMIT;
COMMIT
B=# COMMIT;
COMMIT
A=# SELECT sum(price) FROM shipment WHERE shipment_id IN (501, 502);
  sum
-------
 85.00
(1 row)
```

Regla: entre el 501 y el 502 el cliente paga al menos 90. Cada empleado ve 95 y descuenta 5 a un envío *distinto*.

Write skew en REPEATABLE READ

```
node psql2.mjs write-skew rr

A=# SELECT sum(price) FROM shipment WHERE shipment_id IN (501, 502);
  sum
-------
 95.00
(1 row)
B=# SELECT sum(price) FROM shipment WHERE shipment_id IN (501, 502);
  sum
-------
 95.00
(1 row)
A=# UPDATE shipment SET price = price - 5 WHERE shipment_id = 501;
UPDATE 1
B=# UPDATE shipment SET price = price - 5 WHERE shipment_id = 502;
UPDATE 1
A=# COMMIT;
COMMIT
B=# COMMIT;
COMMIT
A=# SELECT sum(price) FROM shipment WHERE shipment_id IN (501, 502);
  sum
-------
 85.00
(1 row)
```

Regla: entre el 501 y el 502 el cliente paga al menos 90. Cada empleado ve 95 y descuenta 5 a un envío *distinto*.

- Final 85: regla rota, sin error y sin esperas.
- Filas distintas: no hay `w-w` que snapshot isolation detecte.
- `r1(A) r1(B) r2(A) r2(B) w1(A) w2(B)`: `r1(B)–w2(B)` y `r2(A)–w1(A)` cierran un ciclo.

Write skew en `SERIALIZABLE`

```
node psql2.mjs write-skew sr

...
A=# UPDATE shipment SET price = price - 5 WHERE shipment_id = 501;
UPDATE 1
B=# UPDATE shipment SET price = price - 5 WHERE shipment_id = 502;
UPDATE 1
A=# COMMIT;
COMMIT
B=# COMMIT;
ERROR:  could not serialize access due to read/write dependencies among transactions
DETAIL:  Reason code: Canceled on identification as a pivot, during commit attempt.
HINT:  The transaction might succeed if retried.
A=# SELECT sum(price) FROM shipment WHERE shipment_id IN (501, 502);
  sum
-------
 90.00
(1 row)
```

- El error llega en el `COMMIT` y es otro: *read/write dependencies*. Mismo código, `40001`.

Write skew en `SERIALIZABLE`

```
node psql2.mjs write-skew sr

...
A=# UPDATE shipment SET price = price - 5 WHERE shipment_id = 501;
UPDATE 1
B=# UPDATE shipment SET price = price - 5 WHERE shipment_id = 502;
UPDATE 1
A=# COMMIT;
COMMIT
B=# COMMIT;
ERROR:  could not serialize access due to read/write dependencies among transactions
DETAIL:  Reason code: Canceled on identification as a pivot, during commit attempt.
HINT:  The transaction might succeed if retried.
A=# SELECT sum(price) FROM shipment WHERE shipment_id IN (501, 502);
  sum
-------
 90.00
(1 row)
```

- El error llega en el `COMMIT` y es otro: *read/write dependencies*. Mismo código, `40001`.
- Error común: “`REPEATABLE READ` de PostgreSQL = `SERIALIZABLE`”. No: no hay phantoms, pero sí write skew.

`SERIALIZABLE`: SSI, en breve

Serializable snapshot isolation (Cahill, Röhm y Fekete, 2008; PostgreSQL 9.1):

1. Todo lo de `REPEATABLE READ`: snapshot fijo, la primera que actualiza una fila gana.

`SERIALIZABLE`: SSI, en breve

Serializable snapshot isolation (Cahill, Röhm y Fekete, 2008; PostgreSQL 9.1):

1. Todo lo de `REPEATABLE READ`: snapshot fijo, la primera que actualiza una fila gana.
2. Cada lectura deja una marca que *no bloquea* (`SIReadLock` en `pg_locks`); con ellas se ven las dependencias `r-w` entre concurrentes.

`SERIALIZABLE`: SSI, en breve

Serializable snapshot isolation (Cahill, Röhm y Fekete, 2008; PostgreSQL 9.1):

1. Todo lo de `REPEATABLE READ`: snapshot fijo, la primera que actualiza una fila gana.
2. Cada lectura deja una marca que *no bloquea* (`SIReadLock` en `pg_locks`); con ellas se ven las dependencias `r-w` entre concurrentes.
3. Una transacción con una dependencia `r-w` de entrada y otra de salida (un *pivot*) puede cerrar un ciclo: se aborta a alguien con `40001`.

`SERIALIZABLE`: SSI, en breve

Serializable snapshot isolation (Cahill, Röhm y Fekete, 2008; PostgreSQL 9.1):

1. Todo lo de `REPEATABLE READ`: snapshot fijo, la primera que actualiza una fila gana.
2. Cada lectura deja una marca que *no bloquea* (`SIReadLock` en `pg_locks`); con ellas se ven las dependencias `r-w` entre concurrentes.
3. Una transacción con una dependencia `r-w` de entrada y otra de salida (un *pivot*) puede cerrar un ciclo: se aborta a alguien con `40001`.

El trato

Conservador: a veces aborta sin que haya ciclo; nunca deja pasar uno. Todo lo que hace commit es serializable. El precio: la aplicación reintenta. Detalles en la semana 14.

Cómo se implementa el aislamiento (17.9)

- **Locks:** permiso para leer (compartido) o escribir (exclusivo). Con 2PL, conflict serializable. Para phantoms, locks de predicado o de rango del índice.
- **Timestamps:** el resultado debe ser el del orden por timestamps; quien llega tarde aborta.
- **Varias versiones y snapshot isolation:** cada escritura crea una versión; cada transacción lee de su snapshot. Los lectores no esperan.

Cómo se implementa el aislamiento (17.9)

- **Locks:** permiso para leer (compartido) o escribir (exclusivo). Con 2PL, conflict serializable. Para phantoms, locks de predicado o de rango del índice.
- **Timestamps:** el resultado debe ser el del orden por timestamps; quien llega tarde aborta.
- **Varias versiones y snapshot isolation:** cada escritura crea una versión; cada transacción lee de su snapshot. Los lectores no esperan.

Lo que vimos	Técnica
A lee mientras B tiene la fila, sin esperar	varias versiones
El `UPDATE` de B espera al de A	lock de fila
`40001` *concurrent update*	snapshot isolation
`40001` *read/write dependencies*	SSI

SECCIÓN 07

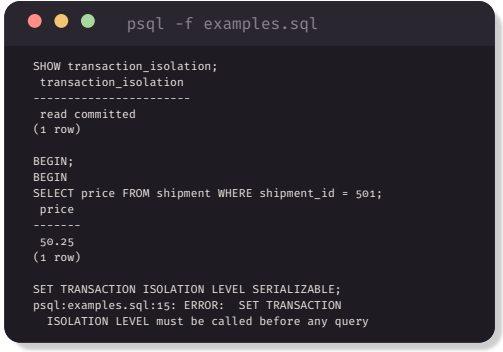
Transacciones en SQL

07



`BEGIN`, `COMMIT`, niveles

- Sin `BEGIN`: **autocommit**, cada sentencia es una transacción (`psql` y `pg`).
- `BEGIN` o `START TRANSACTION` (estándar); `COMMIT`; `ROLLBACK`.
- `BEGIN ISOLATION LEVEL SERIALIZABLE`, o `SET TRANSACTION ...` como primera sentencia.
- Nivel por defecto: `default_transaction_isolation`, `read committed`.



(la última línea, partida a mano)

Un error aborta toda la transacción

psql -f examples.sql

```
BEGIN;
BEGIN
UPDATE shipment SET price = price + 5 WHERE shipment_id = 501;
UPDATE 1
UPDATE shipment SET price = -1 WHERE shipment_id = 502;
psql:examples.sql:22: ERROR:  new row for relation "shipment" violates check constraint "shipment_price_check"
DETAIL:  Failing row contains (502, 364, 966, 1, 15, 4945, 0.90, 1695.00, -1.00, delivered, ...).
SELECT price FROM shipment WHERE shipment_id = 501;
psql:examples.sql:23: ERROR:  current transaction is aborted, commands ignored until end of transaction block
COMMIT;
ROLLBACK
```

- Después del error, todo se ignora (`25P02`) y `COMMIT` responde `ROLLBACK`: el primer `UPDATE` también se deshace.
- Otros motores deshacen solo la sentencia que falló; PostgreSQL no.

Savepoints

psql -f examples.sql

```
BEGIN;
BEGIN
UPDATE shipment SET price = price + 5 WHERE shipment_id = 501;
UPDATE 1
SAVEPOINT before_discount;
SAVEPOINT
UPDATE shipment SET price = -1 WHERE shipment_id = 502;
psql:examples.sql:31: ERROR:  new row for relation "shipment"
  violates check constraint "shipment_price_check"
ROLLBACK TO SAVEPOINT before_discount;
ROLLBACK
SELECT shipment_id, price FROM shipment
  WHERE shipment_id IN (501, 502) ORDER BY shipment_id;
 shipment_id | price
-------------+-------
         501 | 55.25
         502 | 45.26
(2 rows)

ROLLBACK;
ROLLBACK
```

- `ROLLBACK TO SAVEPOINT` deshace lo posterior y saca a la transacción del error.
- El primer `UPDATE` (55.25) sigue.
- No es una transacción anidada: no confirma nada.
- En `psql`: `\set ON_ERROR_ROLLBACK interactive`.

(sin `DETAIL` y con líneas partidas, para que entre)

SECCIÓN 08

Reintentar

08

Qué se reintenta

SQLSTATE	Nombre	Cuándo
`40001`	`serialization_failure`	RR o `SERIALIZABLE`: misma fila, o un ciclo posible
`40P01`	`deadlock_detected`	cualquier nivel: se esperan mutuamente (semana 14)

- Una restricción violada o un error de sintaxis no se arreglan reintentando.

Qué se reintenta

SQLSTATE	Nombre	Cuándo
`40001`	`serialization_failure`	RR o `SERIALIZABLE`: misma fila, o un ciclo posible
`40P01`	`deadlock_detected`	cualquier nivel: se esperan mutuamente (semana 14)

- Una restricción violada o un error de sintaxis no se arreglan reintentando.
- Se reintenta **la transacción completa**, no la sentencia que falló: leyó de un snapshot que ya no vale y decidió con él.

El reintento a mano

```
node psql2.mjs retry sr

...
B=# UPDATE shipment SET price = 40 WHERE shipment_id = 501;
-- B espera
A=# COMMIT;
COMMIT
 -- B continúa
ERROR:  could not serialize access due to concurrent update
B=# ROLLBACK;
ROLLBACK
B=# BEGIN ISOLATION LEVEL SERIALIZABLE;
BEGIN
B=# SELECT price FROM shipment WHERE shipment_id = 501;
 price
-------
 55.00
(1 row)
B=# UPDATE shipment SET price = 45 WHERE shipment_id = 501;
UPDATE 1
B=# COMMIT;
COMMIT
```

1. `ROLLBACK`.
2. `BEGIN` otra vez: snapshot nuevo.
3. Volver a leer: ahora 55.00.
4. Volver a calcular y escribir: 45.

Resultado: el del orden serial “A, después B”.

Reintentos en Node.js

el ciclo, sobre un cliente

```
for (let attempt = 1; ; attempt++) {
  try {
    await client.query(
      'BEGIN ISOLATION LEVEL SERIALIZABLE');
    // lecturas, cálculos y escrituras
    await client.query('COMMIT');
    break;
  } catch (err) {
    await client.query('ROLLBACK');
    const retry = ['40001', '40P01'].includes(err.code);
    if (!retry || attempt === MAX) throw err;
    // esperar unos ms al azar
  }
}
```

Reintentos en Node.js

el ciclo, sobre un cliente

```
for (let attempt = 1; ; attempt++) {
  try {
    await client.query(
      'BEGIN ISOLATION LEVEL SERIALIZABLE');
    // lecturas, cálculos y escrituras
    await client.query('COMMIT');
    break;
  } catch (err) {
    await client.query('ROLLBACK');
    const retry = ['40001', '40P01'].includes(err.code);
    if (!retry || attempt === MAX) throw err;
    // esperar unos ms al azar
  }
}
```

1. **Todo** adentro, también las lecturas.

Reintentos en Node.js

el ciclo, sobre un cliente

```
for (let attempt = 1; ; attempt++) {
  try {
    await client.query(
      'BEGIN ISOLATION LEVEL SERIALIZABLE');
    // lecturas, cálculos y escrituras
    await client.query('COMMIT');
    break;
  } catch (err) {
    await client.query('ROLLBACK');
    const retry = ['40001', '40P01'].includes(err.code);
    if (!retry || attempt === MAX) throw err;
    // esperar unos ms al azar
  }
}
```

1. **Todo** adentro, también las lecturas.
2. `ROLLBACK` ante *cualquier* error.

Reintentos en Node.js

el ciclo, sobre un cliente

```
for (let attempt = 1; ; attempt++) {
  try {
    await client.query(
      'BEGIN ISOLATION LEVEL SERIALIZABLE');
    // lecturas, cálculos y escrituras
    await client.query('COMMIT');
    break;
  } catch (err) {
    await client.query('ROLLBACK');
    const retry = ['40001', '40P01'].includes(err.code);
    if (!retry || attempt === MAX) throw err;
    // esperar unos ms al azar
  }
}
```

1. **Todo** adentro, también las lecturas.
2. `ROLLBACK` ante *cualquier* error.
3. Límite de intentos y espera al azar.

Reintentos en Node.js

el ciclo, sobre un cliente

```
for (let attempt = 1; ; attempt++) {
  try {
    await client.query(
      'BEGIN ISOLATION LEVEL SERIALIZABLE');
    // lecturas, cálculos y escrituras
    await client.query('COMMIT');
    break;
  } catch (err) {
    await client.query('ROLLBACK');
    const retry = ['40001', '40P01'].includes(err.code);
    if (!retry || attempt === MAX) throw err;
    // esperar unos ms al azar
  }
}
```

1. **Todo** adentro, también las lecturas.
2. `ROLLBACK` ante *cualquier* error.
3. Límite de intentos y espera al azar.
4. SMS y pagos **después** del ciclo.

Reintentos en Node.js

el ciclo, sobre un cliente

```
for (let attempt = 1; ; attempt++) {
  try {
    await client.query(
      'BEGIN ISOLATION LEVEL SERIALIZABLE');
    // lecturas, cálculos y escrituras
    await client.query('COMMIT');
    break;
  } catch (err) {
    await client.query('ROLLBACK');
    const retry = ['40001', '40P01'].includes(err.code);
    if (!retry || attempt === MAX) throw err;
    // esperar unos ms al azar
  }
}
```

1. **Todo** adentro, también las lecturas.
2. `ROLLBACK` ante *cualquier* error.
3. Límite de intentos y espera al azar.
4. SMS y pagos **después** del ciclo.

Semana 5

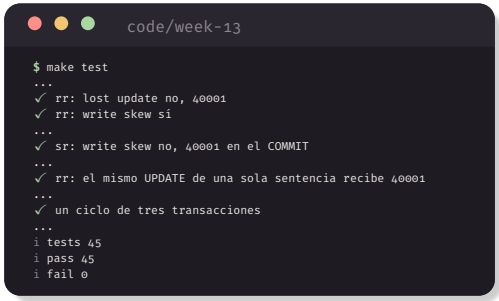
Un solo cliente de `pool.connect()`; nunca `pool.query('BEGIN')`.

El código de la semana

- `precedence.mjs`: conflictos, grafo, orden serial o ciclo; recuperable y cascadeless.
- `graph.mjs`: lo muestra para un schedule.
- `scenarios.mjs`, `sessions.mjs`, `psql2.mjs`: los schedules de dos sesiones, con `pg` o con dos `psql`.
- `*.test.mjs`: cada anomalía en cada nivel contra PostgreSQL.

¿Espera o terminó?

Una tercera conexión pregunta `pg_blocking_pids(pid)`.



Resumen

- **Transacción**: entera o nada, como si estuviera sola. ACID: A y D, recuperación con el log; I, control de concurrencia; C, el programador.
- **Schedule**: el intercalado. **Serializable**: equivale a *algún* serial.
- **Conflicto**: transacciones distintas, mismo ítem, al menos una escritura. Conflict serializable $\iff$ grafo de precedencia acíclico; orden topológico = serial.
- View serializability: acepta escrituras ciegas; NP-completo.
- **Recuperable**: quien lee de Ti confirma después de Ti. **Cascadeless**: solo lee lo confirmado.
- Niveles SQL por anomalías permitidas: dirty read, non-repeatable read, phantom. Además: lost update, write skew.
- PostgreSQL: RU = RC (snapshot por sentencia); RR = snapshot isolation (sin phantoms, con write skew); `SERIALIZABLE` = SSI. Los fuertes abortan con `40001`.
- Se reintenta la transacción completa: `ROLLBACK`, un solo cliente, límite y azar.

Para la próxima clase

- Laboratorio: anomalías con dos sesiones de `psql` y una transacción con reintentos en Node.js.
- Leer Silberschatz, secs. 18.1 a 18.4 (protocolos con locks, deadlocks, granularidad múltiple) y la nota 12 de CS186.
- Correr `make test` en `code/week-13` y leer los apuntes de la semana.

Pregunta

En el lost update de `READ COMMITTED`, el `UPDATE` de B esperó un lock y aun así el resultado no fue serial. ¿Qué locks tendría que pedir cada transacción, y *cuándo* soltarlos, para que todo schedule fuera serializable?