

Bases de Datos

Control de concurrencia: locks, deadlocks y MVCC

Transacciones y sistemas distribuidos

Gerson Garrido

Semana 14

UNIVERSIDAD NACIONAL DE INGENIERÍA

Hoy

Sesión A: locks y deadlocks

1. Locks S y X, y por qué no bastan
2. Two-phase locking: básico, strict, rigorous
3. El lock manager por dentro
4. Deadlocks: prevención y detección
5. Granularidad múltiple y phantoms

Sesión B: MVCC

6. Esquemas multiversión y snapshot isolation
7. MVCC en PostgreSQL: `xmin`, `xmax`, `VACUUM`
8. Write skew y cómo evitarlo
9. (Opcional) Timestamps y validación

Lectura

Silberschatz, secs. 18.1–18.4 (sesión A) y 18.7–18.8 (sesión B); opcionales, 18.5–18.6. CS186, *Transactions* & *Concurrency II*. PostgreSQL, *Concurrency Control* y *System Columns*. Además: CMU 15-445 (Fall 2025), *Two-Phase Locking* y *Multi-Version Concurrency Control* (opcional, *Timestamp Ordering*); PostgreSQL, *Explicit Locking*, `pg_locks` y *Routine Vacuuming*.

SECCIÓN 01

Locks

01



Del qué al cómo

- Semana 13: qué queremos. Un schedule conflict serializable, el grafo de precedencia, los niveles de aislamiento.
- Pero el DBMS no puede esperar al final del schedule para revisarlo.
- Tiene que decidir en el momento: dejar pasar, hacer esperar o abortar.

Pesimista: locks

Pedir permiso antes de tocar un ítem; si otro lo tiene, esperar. Two-phase locking. Trae deadlocks.

Multiversión: MVCC

Guardar versiones viejas para que los lectores no esperen. Snapshot isolation. Trae el write skew.

Shared y exclusive (sec. 18.1.1)

- **S** *(shared)*: `lock-S(Q)`, permite leer Q.
- **X** *(exclusive)*: `lock-X(Q)`, permite leer y escribir Q.
- `unlock(Q)` lo suelta.
- Si el modo pedido no es compatible con un lock ya concedido a otra transacción, la que pide espera.

	S	X
S	✓	
X		

matriz de compatibilidad

Definición

Ti puede recibir un lock en modo A sobre Q si y solo si `comp(A, B)` vale para todo lock en modo B que otra Tj tenga sobre Q. Con S y X, solo `comp(S, S)`.

- Con Sistemas Operativos: X es un mutex por ítem; S/X, un readers-writer lock.

Tener locks no basta

A = precio del envío 501, B = del 502. T2 pasa 10 soles de A a B; T1 suma A + B.

T1	T2
`lock-S(A) r(A)`	
`unlock(A)`	
	`lock-X(A) w(A)`
	`lock-X(B) w(B)`
	`unlock(A) unlock(B)`
`lock-S(B) r(B)`	
`unlock(B)`	

- Ningún lock se violó.
- T1 suma el A *de antes* y el B *de después*: 10 soles de más.
- Grafo: T1 $\rightarrow$ T2 (A) y T2 $\rightarrow$ T1 (B). Ciclo.
- T1 soltó un lock y después pidió otro.

Idea clave

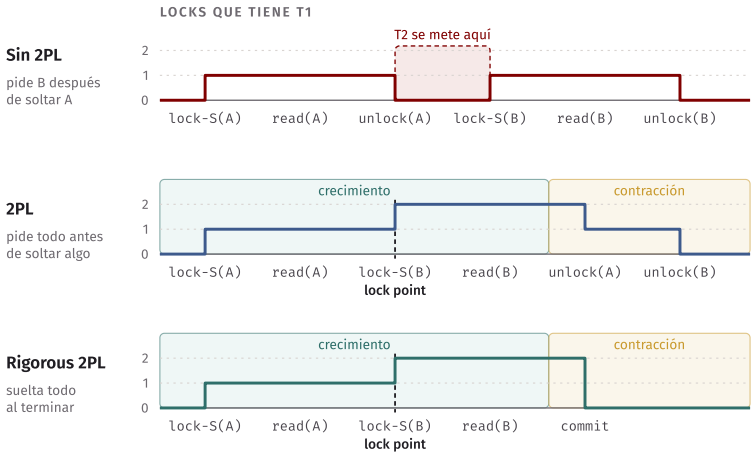
Para la serializabilidad hace falta un protocolo: cuándo se pide y cuándo se suelta cada lock.

Two-phase locking (sec. 18.1.3)

1. **Crecimiento**: puede pedir locks, no soltar.
2. **Contracción**: puede soltar locks, no pedir.

- Al soltar el primero, no hay vuelta atrás.
- **Lock point**: el momento en que obtiene su último lock.
- T1 en 2PL: `lock-S(A) r(A) lock-S(B) r(B) unlock(A) unlock(B)`.

Los locks de T1, en tres versiones



2PL garantiza conflict serializability

Teorema

Si todas las transacciones siguen 2PL, el schedule es conflict serializable; el orden de los lock points es un orden serial equivalente.

- Arista Ti $\rightarrow$ Tj: conflicto sobre Q, Ti primero, locks incompatibles.
- Ti soltó Q antes de que Tj lo obtuviera.
- lockpoint(Ti) $<$ $\texttt{unlock}_i$(Q) $<$ $\texttt{lock}_j$(Q) $\leq$ lockpoint(Tj).
- Un ciclo daría lockpoint(T1) $<$ lockpoint(T1). □

Solo en un sentido

Hay schedules conflict serializables que 2PL no produce. Es suficiente, no necesario. A cambio, no hay que saber nada de antemano.

Rollbacks en cascada: strict y rigorous 2PL

T1	T2
`lock-X(A) w(A)`	
`lock-S(B) r(B)`	
`unlock(A)`	
	`lock-S(A) r(A)`
abort	

Las dos cumplen 2PL, y T2 leyó algo que nunca hará commit.

Strict 2PL

2PL + los locks X se guardan hasta el commit o abort. Schedules cascadeless.

Rigorous 2PL

2PL + todos los locks hasta el final. Cascadeless y serializado en el orden de los commits.

- Strict no da el orden de commits: T1 lee A y suelta su S; T2 escribe A y hace commit; T1 hace commit después. Arista T1 $\rightarrow$ T2, commits T2, T1.
- PostgreSQL guarda los locks de fila y de tabla hasta el fin de la transacción.

Conversiones y locks automáticos

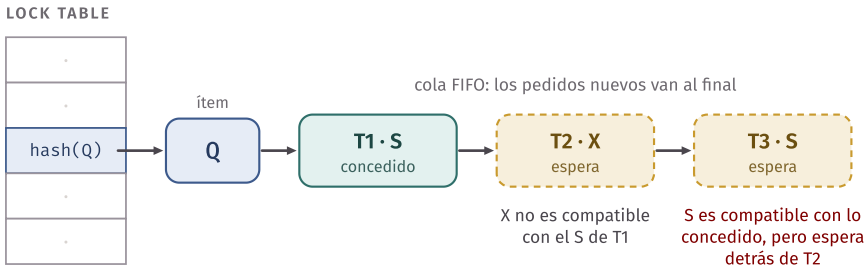
- Leer con S y después querer escribir: `upgrade(Q)`, de S a X sin soltar el S.
- `downgrade(Q)`: de X a S.
- Con 2PL: upgrades solo en crecimiento, downgrades solo en contracción.

Lo que hace el DBMS solo (sec. 18.1.4)

`read(Q)` pide `lock-S(Q)`; `write(Q)` pide `upgrade(Q)` o `lock-X(Q)`; todo se suelta al commit o abort. Eso es rigorous 2PL.

- Protocolo de árbol (sec. 18.1.5): sin 2PL, pero con un orden de acceso conocido. Sin deadlocks. Es la idea del *lock crabbing* en un B+ tree.

El lock manager por dentro



Si T3 pasara adelante, y luego T4 y T5 con más S, siempre habría un lector y el X de T2 no se concedería nunca: starvation.

El lock manager: las reglas

- Una entrada por ítem: concedidos y en espera.
- Un pedido se concede si es compatible con los concedidos y nadie espera antes.
- Al soltar, se revisa la cola en orden.
- Al abortar, se borran todos los pedidos de la transacción.

¿Por qué *nadie espera antes*?

T1 tiene S, T2 espera X. Si se le concede S a cada lector nuevo, T2 no avanza nunca: **starvation**. Con la cola FIFO, el lector nuevo espera detrás de T2.

Esto es P5

P5: el lock manager de `quipu`

Locks S y X sobre registros, strict 2PL, upgrades, cola de espera por recurso y detección de deadlocks con un waits-for graph. Se publica hoy; se entrega en la semana 16.

- El caso delicado: el upgrade. Si se pone al final de la cola, la transacción puede quedar esperándose a sí misma.
- El código de esta semana no es un lock manager: usa los locks de PostgreSQL y trae una función genérica para buscar ciclos.

PostgreSQL: los locks de una lectura y de una escritura

node locks.mjs

```
A: BEGIN
A: SELECT price FROM shipment WHERE shipment_id = 501
 session | locktype |   relation    | tuple | xid |      mode       | granted
---------+----------+---------------+-------+-----+-----------------+---------
 A       | relation | shipment      |       |     | AccessShareLock | t
 A       | relation | shipment_pkey |       |     | AccessShareLock | t
(2 rows)

A: UPDATE shipment SET price = price + 1 WHERE shipment_id = 501
 session |   locktype    |   relation    | tuple |  xid  |       mode       | granted
---------+---------------+---------------+-------+-------+------------------+---------
 A       | relation      | shipment      |       |       | AccessShareLock  | t
 A       | relation      | shipment      |       |       | RowExclusiveLock | t
 A       | relation      | shipment_pkey |       |       | AccessShareLock  | t
 A       | relation      | shipment_pkey |       |       | RowExclusiveLock | t
 A       | transactionid |               |       | 25434 | ExclusiveLock    | t
(5 rows)
```

- Locks de tabla, ninguno de fila. Y un X sobre el propio xid de A.
- El lock de la fila 501 está en la tupla (`xmax`), no en la lock table.

Una espera: `ShareLock` sobre el xid de otra

node locks.mjs

```
B: UPDATE shipment SET price = price + 1 WHERE shipment_id = 501   -- espera
 session |   locktype    |   relation    | tuple  |  xid  |       mode       | granted
---------+---------------+---------------+--------+-------+------------------+---------
 A       | transactionid |               |        | 25434 | ExclusiveLock    | t
 B       | transactionid |               |        | 25435 | ExclusiveLock    | t
 B       | tuple         | shipment      | 147,65 |       | ExclusiveLock    | t
 B       | transactionid |               |        | 25434 | ShareLock        | f
 session | pid  | blocked_by | wait_event_type |        state
---------+------+------------+-----------------+---------------------
 A       | 3908 | {}         | Client          | idle in transaction
 B       | 3909 | {3908}     | Lock            | active
```

(solo algunas filas de la salida)

- B espera que A termine: pide S sobre el xid 25434, que A tiene en X.
- El lock `tuple` es el lugar de B en la fila de espera de esa fila.
- Estas filas aparecen solo cuando alguien espera.

SECCIÓN 02

Deadlocks

02



Dos maneras de trabarse

Orden cruzado

T1	T2
`lock-X(501)`	
	`lock-X(502)`
`lock-X(502)` espera	
	`lock-X(501)` espera

Upgrade

T1	T2
`lock-S(501)`	
	`lock-S(501)`
`upgrade(501)` espera	
	`upgrade(501)` espera

Una sola fila. Por eso existen `FOR UPDATE` y los update locks (U).

Deadlock (sec. 18.2)

Un conjunto de transacciones en el que cada una espera a otra del conjunto.

2PL provoca deadlocks

- Las condiciones de Coffman, en 2PL por diseño: exclusión mutua (X), hold and wait (la fase de crecimiento), no preemption, espera circular.
- Solo 2PL conservador (todos los locks al empezar) los evita, y casi nunca se puede: no se sabe de antemano qué filas se van a tocar.

Error común

"2PL evita deadlocks." Es al revés.

Error común

Confundir 2PL (concurrencia en un servidor) con 2PC (commit atómico entre varios servidores, semana 16). Solo comparten las "dos fases".

Prevención (sec. 18.2.1)

- **Orden fijo**: todos piden locks en orden de `shipment_id`. Lo más útil en una aplicación.
- **Timestamps**: TS(Ti) al empezar; menor = más antigua. Ti pide algo que tiene Tj:

	Ti más antigua	**Ti más joven**
Wait-die (no preemptivo)	Ti espera	Ti muere (rollback)
Wound-wait (preemptivo)	Ti hiere a Tj (rollback de Tj)	Ti espera

- Todas las esperas van en un solo sentido del tiempo: no hay ciclo.
- Se reinicia con su timestamp original: algún día es la más antigua. Sin starvation.
- **Timeouts**: simples, pero adivinan (`lock_timeout` en PostgreSQL).

Wait-die y wound-wait con dos transacciones

T1 tiene timestamp 3; T2 tiene 8.

T2 tiene X sobre 501; T1 lo pide

- Wait-die: T1 es más antigua, espera.
- Wound-wait: T1 hiere a T2; T2 hace rollback.

T1 tiene X sobre 501; T2 lo pide

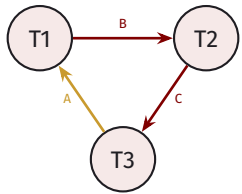
- Wait-die: T2 es más joven, muere.
- Wound-wait: T2 espera.

El defecto de los dos

Hacen rollback aunque no haya deadlock, solo por el orden de los timestamps. En wait-die, una joven puede morir varias veces por el mismo ítem.

Detección: el waits-for graph (sec. 18.2.3)

- Un nodo por transacción; Ti → Tj si Ti espera un lock que tiene Tj.
- T1 tiene X(A) y espera S(B); T2 tiene X(B) y espera X(C); T3 tiene X(C).
- T3 pide S(A): se cierra el ciclo.



Idea clave

Hay deadlock si y solo si el waits-for graph tiene un ciclo. Una cadena de esperas termina; un ciclo, no.

Recuperarse de un deadlock

1. **Víctima**: la que cueste menos. Cuánto trabajó y le falta, cuántos ítems usó y necesita, cuántas más habría que deshacer.
2. **Rollback**: total, o parcial hasta el lock que rompe el ciclo (los savepoints son la versión para el programador).
3. **Starvation**: sumar al costo cuántas veces ya fue víctima.

- ¿Cada cuánto buscar ciclos? En cada espera (caro), cada cierto tiempo, o cuando alguien lleva un rato esperando (PostgreSQL).

Buscar el ciclo: `findCycle`

waitsfor.mjs

```
1  import { findCycle } from './waitsfor.mjs';
2
3  // Grafo de precedencia de la semana 13
4  findCycle([['T1', 'T3'], ['T2', 'T3'], ['T2', 'T1']]);
5  // -> { acyclic: true, order: ['T2', 'T1', 'T3'] }
6
7  // Waits-for graph de la diapositiva anterior, con T3 esperando a T1
8  findCycle([['T1', 'T2'], ['T2', 'T3'], ['T3', 'T1']]);
9  // -> { acyclic: false, cycle: ['T1', 'T2', 'T3'] }
```

- DFS con tres colores: una arista hacia un nodo gris (en el camino actual) cierra un ciclo.
- Sin ciclo, el orden inverso de terminación es un orden topológico.
- No sabe nada de locks: en P5, el grafo lo arman ustedes desde su lock table.

El deadlock de `deadlock.mjs`




```
B: ERROR:  deadlock detected (40P01)
```

a los 2 s (deadlock_timeout de B), B encuentra el ciclo y aborta; A continúa

Un deadlock real: las dos esperando

node deadlock.mjs

```
A: UPDATE shipment SET price = price + 1 WHERE shipment_id = 501
B: UPDATE shipment SET price = price + 1 WHERE shipment_id = 502
A: UPDATE shipment SET price = price + 1 WHERE shipment_id = 502   -- espera
B: UPDATE shipment SET price = price + 1 WHERE shipment_id = 501   -- espera

 session |   locktype    |   relation    | tuple  |  xid  |       mode       | granted
---------+---------------+---------------+--------+-------+------------------+---------
 A       | transactionid |               |        | 25437 | ExclusiveLock    | t
 A       | tuple         | shipment      | 6,15   |       | ExclusiveLock    | t
 A       | transactionid |               |        | 25438 | ShareLock        | f
 B       | transactionid |               |        | 25438 | ExclusiveLock    | t
 B       | tuple         | shipment      | 147,65 |       | ExclusiveLock    | t
 B       | transactionid |               |        | 25437 | ShareLock        | f

 session | pid  | blocked_by | wait_event_type | state
---------+------+------------+-----------------+--------
 A       | 3936 | {3937}     | Lock            | active
 B       | 3937 | {3936}     | Lock            | active
(2 rows)
findCycle: {"acyclic":false,"cycle":[3936,3937]}
```

(sin los locks de tabla) Las dos filas con `granted = f` son las dos aristas: A $\rightarrow$ B y B $\rightarrow$ A.

El error `40P01`

```
node deadlock.mjs

B: ERROR:  deadlock detected  (código 40P01)
DETAIL:  Process 3937 waits for ShareLock on transaction 25437; blocked by process 3936.
         Process 3936 waits for ShareLock on transaction 25438; blocked by process 3937.
A: UPDATE 1   -- continúa
B: ROLLBACK
A: ROLLBACK
```

- El `DETAIL` describe el ciclo: procesos y transacciones.
- B queda abortada y suelta sus locks en ese momento; A sigue.
- B solo puede hacer `ROLLBACK`.

Detección perezosa: `deadlock_timeout`

- Al esperar, la sesión pone un temporizador (1 s por defecto). Si vence, esa sesión busca un ciclo que pase por ella y, si lo hay, se aborta a sí misma. Sin elección por costo.
- Arriba: A 4 s, B 2 s; víctima B. Al revés (A 1 s, B 4 s):

```
node deadlock.mjs cross 1s 4s

A: ERROR:  deadlock detected  (código 40P01)
DETAIL:  Process 5518 waits for ShareLock on transaction 27055; blocked by process 5519.
         Process 5519 waits for ShareLock on transaction 27054; blocked by process 5518.
B: UPDATE 1   -- continúa
```

Por qué así

Buscar ciclos es caro y los deadlocks son raros; un deadlock dura $\geq$ `deadlock_timeout`.

El deadlock por upgrade, en vivo

```
node deadlock.mjs upgrade

A: SELECT price FROM shipment WHERE shipment_id = 501 FOR SHARE
B: SELECT price FROM shipment WHERE shipment_id = 501 FOR SHARE
A: UPDATE shipment SET price = price + 1 WHERE shipment_id = 501   -- espera
B: UPDATE shipment SET price = price + 1 WHERE shipment_id = 501   -- espera

 session | pid  | blocked_by | wait_event_type | state
---------+------+------------+-----------------+--------
 A       | 3958 | {3959}     | Lock            | active
 B       | 3959 | {3958}     | Lock            | active
(2 rows)

B: ERROR:  deadlock detected  (código 40P01)
DETAIL:  Process 3959 waits for ShareLock on transaction 25440; blocked by process 3958.
         Process 3958 waits for ShareLock on transaction 25441; blocked by process 3959.
```

- `FOR SHARE` es el `lock-S` de una fila. Las dos tienen S; las dos quieren X.
- Si se piensa escribir, leer con `FOR UPDATE`: la segunda espera en la lectura.

En la aplicación

- Ante `40P01`, igual que ante `40001`: `ROLLBACK` y reintentar la transacción completa (la función de reintento de la semana 13).
- Para que sean raros: locks en un orden fijo.

```sql
SELECT shipment_id
FROM shipment
WHERE shipment_id = ANY ($1)
ORDER BY shipment_id
FOR UPDATE;
```

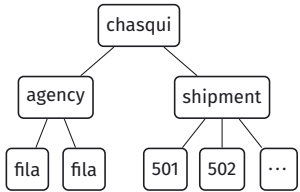
SECCIÓN 03

Granularidad múltiple

03

Filas o tablas (sec. 18.3)

- Recalcular 12 000 precios con locks de fila: 12 000 locks.
- Cambiar un envío con un lock de tabla: nadie más toca `shipment`.
- Que cada transacción elija la granularidad.
- Un lock S o X sobre un nodo bloquea implícitamente su subárbol.



El problema

T1 tiene X sobre `shipment`. T2 quiere S sobre el envío 501: ¿cómo sabe del X implícito? T3 quiere X sobre la tabla: ¿revisa cada fila?

Locks de intención

- **IS**: voy a pedir S más abajo.
- **IX**: voy a pedir X (o S) más abajo.
- **SIX**: S sobre todo el nodo + IX.
- Se marcan los ancestros antes de bloquear un nodo.

IX con IX: dos escrituras en filas distintas conviven. IX con S: no.

	IS	IX	S	SIX	X
IS	✓	✓	✓	✓	
IX	✓	✓			
S	✓		✓		
SIX	✓				
X					

El protocolo

1. Respetar la matriz.
2. La raíz primero, en cualquier modo.
3. S o IS en Q: el padre en IX o IS.
4. X, SIX o IX en Q: el padre en IX o SIX.
5. 2PL.
6. Soltar Q solo si no tiene hijos bloqueados.

De la raíz a las hojas para pedir; de las hojas a la raíz para soltar.

- Leer el 501: IS, IS, S.
- Actualizar el 501: IX, IX, X.
- Leer toda `agency`: IS, S. Ningún lock de fila.
- Leer todos los envíos y corregir algunos: IX, SIX, X en cada fila que cambia.

Lock escalation

Muchos locks de fila en una tabla se cambian por uno de tabla: menos memoria, menos concurrencia.

PostgreSQL: ocho modos de tabla, matriz medida

```
node locks.mjs matrix
```

tiene \ pide	AS	RS	RE	SUE	S	SRE	E	AE
ACCESS SHARE	ok	ok	ok	ok	ok	ok	ok	X
ROW SHARE	ok	ok	ok	ok	ok	ok	X	X
ROW EXCLUSIVE	ok	ok	ok	ok	X	X	X	X
SHARE UPDATE EXCLUSIVE	ok	ok	ok	X	X	X	X	X
SHARE	ok	ok	X	X	ok	X	X	X
SHARE ROW EXCLUSIVE	ok	ok	X	X	X	X	X	X
EXCLUSIVE	ok	X	X	X	X	X	X	X
ACCESS EXCLUSIVE	X	X	X	X	X	X	X	X

- A toma un modo sobre `agency`; B pide cada modo con `NOWAIT` (falla con `55P03` en vez de esperar).
- Es la tabla *Conflicting Lock Modes* de la documentación; la prueba la compara entrada por entrada.

Cada modo y su papel

Modo	Lo toma	Papel
`ACCESS SHARE`	`SELECT`	como IS
`ROW SHARE`	`SELECT ... FOR UPDATE/SHARE`	intención de bloquear filas
`ROW EXCLUSIVE`	`INSERT`, `UPDATE`, `DELETE`	como IX
`SHARE UPDATE EXCLUSIVE`	`VACUUM`, `ANALYZE`	choca consigo mismo
`SHARE`	`CREATE INDEX`	como S
`SHARE ROW EXCLUSIVE`	`CREATE TRIGGER`	como SIX
`EXCLUSIVE`	`REFRESH MAT. VIEW CONCURRENTLY`	como X, deja leer
`ACCESS EXCLUSIVE`	`DROP`, `TRUNCATE`, `ALTER`	X

- Los locks de fila no están en la lock table: no hay lock escalation.
- Mismos nombres para todo: el `ShareLock` de una tabla y el de un xid son el mismo modo sobre objetos distintos.

IX contra S, en vivo

node locks.mjs

```
A: UPDATE shipment SET price = price + 1 WHERE shipment_id = 501
B: BEGIN
B: LOCK TABLE shipment IN SHARE MODE   -- espera
 session |   locktype    |   relation    | tuple |  xid  |       mode       | granted
---------+---------------+---------------+-------+-------+------------------+---------
 A       | relation      | shipment      |       |       | RowExclusiveLock | t
 A       | relation      | shipment_pkey |       |       | RowExclusiveLock | t
 A       | transactionid |               |       | 25436 | ExclusiveLock    | t
 B       | relation      | shipment      |       |       | ShareLock        | f
(4 rows)

A: ROLLBACK
B: LOCK TABLE   -- continúa
```

`ROW EXCLUSIVE` (IX) choca con `SHARE` (S): quien congela la tabla espera a quien escribe una fila.

Los locks de fila de PostgreSQL

- Cuatro modos, de más fuerte a más débil:
  - `FOR UPDATE`: `DELETE` y `UPDATE` de una columna con índice único.
  - `FOR NO KEY UPDATE`: el resto de los `UPDATE`.
  - `FOR SHARE`.
  - `FOR KEY SHARE`: la verificación de una clave foránea.
- Insertar un envío bloquea su agencia con `FOR KEY SHARE`: nadie la borra ni cambia su `name`, pero se puede cambiar su `city`.

El `xmax` de la semana 9

Después de `chasqui-load`, las filas referenciadas tienen `xmax = xmin`: el lock `FOR KEY SHARE` que dejó la carga.

SECCIÓN 04

Phantoms

04

El phantom (sec. 18.4)

- `delete(Q)` necesita X sobre Q; `insert(Q)` da X sobre la fila nueva.
- Eso no basta:

T1

```
1  SELECT sum(weight_kg) FROM shipment WHERE trip_id = 12;
```

- T2 inserta un envío con `trip_id = 12` y hace commit.
- T1 tiene S sobre lo que leyó; T2, X sobre una fila que no existía. Ningún lock choca.
- Si T1 vuelve a sumar, el resultado cambia: la fila nueva es un **phantom**.

> **Idea clave**
>
> El conflicto es sobre un predicado (`trip_id = 12`), no sobre una fila.

Tres soluciones

1. **Lock de tabla**: funciona, pero un reporte detiene todas las inserciones.
2. **Index locking**: quien busca `trip_id = 12` bloquea en S las hojas del índice que recorre; quien inserta, en X las que modifica. Chocan en la hoja del 12.
   - **Next-key locking**: las claves leídas y la siguiente; cubre el hueco.
3. **Predicate locking**: bloquear el predicado. Exacto, pero caro de verificar.

En PostgreSQL

`REPEATABLE READ` no tiene phantoms: el snapshot es fijo (más que lo que pide el estándar). `SERIALIZABLE` usa locks de predicado aproximados que no bloquean: `SIReadLock`.

SECCIÓN 05

MVCC

05



Que los lectores no esperen

- Con 2PL, un reporte espera a quien modifica, y al revés.
- Pero al reporte le bastaría el valor de antes.

MVCC (sec. 18.7)

Cada `write(Q)` crea una versión nueva de Q; cada `read(Q)` elige la versión que le corresponde a la transacción. Las lecturas no esperan ni hacen esperar.

- La pregunta: qué versión le toca a cada lectura.
- Y quién borra las versiones que ya nadie necesita (garbage collection).

Dos esquemas del libro

Multiversion timestamp ordering

- Cada versión Qk: W-timestamp y R-timestamp.
- Qk = la versión con mayor W-ts $\leq$ TS(Ti).
- `read`: devuelve Qk. Nunca falla.
- `write`: si TS(Ti) $<$ R-ts(Qk), rollback; si TS(Ti) = W-ts(Qk), sobrescribe; si no, versión nueva.
- Puede no ser recuperable.

Multiversion 2PL

- Actualizaciones: rigorous 2PL, versiones nuevas.
- Solo lectura: sin locks; TS = `ts-counter` al empezar; leen la versión con mayor timestamp $\leq$ el suyo.
- Al commit: versiones con `ts-counter + 1`, se incrementa.
- Recuperable y cascadeless.

Snapshot isolation (sec. 18.8)

- Cada transacción lee de un **snapshot**: lo que había hecho commit cuando empezó. Sus escrituras son privadas hasta el commit.
- Si dos concurrentes escriben el mismo ítem, solo una hace commit.

First committer wins

Al hacer commit, si una concurrente ya hizo commit de lo mismo, abortar.

First updater wins

Al escribir, lock; si lo tiene otra, esperar; si ella hace commit, abortar. PostgreSQL: `40001`.

Error común

"Snapshot isolation es serializable." No lo es: permite el **write skew**.

SECCIÓN 06

MVCC en PostgreSQL

06



`xmin`, `xmax` y `ctid`

```
node mvcc.mjs versions

A: SELECT ctid, xmin, xmax, price FROM shipment WHERE shipment_id = 501
   ctid=(6,14) xmin=25380 xmax=25380 price=50.25
A: BEGIN
A: UPDATE shipment SET price = price + 1 WHERE shipment_id = 501
   (xid de A: 25428)
A: SELECT ctid, xmin, xmax, price FROM shipment WHERE shipment_id = 501
   ctid=(147,64) xmin=25428 xmax=0 price=51.25   -- A ve su versión nueva
B: SELECT ctid, xmin, xmax, price FROM shipment WHERE shipment_id = 501
   ctid=(6,14) xmin=25380 xmax=25428 price=50.25   -- B ve la versión vieja
B: SELECT pg_current_snapshot();
   25369:25428:25369
A: COMMIT
B: SELECT ctid, xmin, xmax, price FROM shipment WHERE shipment_id = 501
   ctid=(147,64) xmin=25428 xmax=0 price=51.25   -- ahora B ve la nueva
```

- `xmin`: quién creó la versión. `xmax`: quién la borró, la reemplazó o la bloqueó.
- El estado de cada xid (en curso, commit, abortada) está en `pg_xact`.
- B no esperó a nadie.

Dos versiones del envío 501



A hace commit: la siguiente sentencia de B toma otro snapshot y ve la nueva

Un lock de fila es un `xmax`

```
node mvcc.mjs versions
```

```
A: SELECT price FROM shipment WHERE shipment_id = 501 FOR UPDATE
   (xid de A: 25429)
B: SELECT ctid, xmin, xmax, price FROM shipment WHERE shipment_id = 501
   ctid=(147,64) xmin=25428 xmax=25429 price=51.25   -- mismo ctid, xmax es A
A: ROLLBACK
B: SELECT ctid, xmin, xmax, price FROM shipment WHERE shipment_id = 501
   ctid=(147,64) xmin=25428 xmax=25429 price=51.25   -- xmax sigue ahí, pero A abortó
```

- Sin versión nueva: `xmax` con una marca de “solo lock”.
- El rollback no lo borra: quien lo lea ve en `pg_xact` que 25429 abortó.

Snapshots y visibilidad

`pg_current_snapshot()` = `xmin:xmax:xip`

- `xmin`: el menor xid en curso.
- `xmax`: uno más que el mayor xid terminado. De ahí para arriba, en curso.
- `xip`: los en curso entre los dos.
- `25369:25428:25369`: A (25428) no se ve; 25369 es otra sesión del servidor.

Visible, simplificado

(1) La creó ella, o su `xmin` hizo commit y no estaba en curso en el snapshot; y (2) `xmax` es 0, solo un lock, abortó o estaba en curso en el snapshot.

- `READ COMMITTED`: un snapshot por sentencia. Si la fila a actualizar cambió, espera, toma la versión nueva y reevalúa el `WHERE`.
- `REPEATABLE READ` y `SERIALIZABLE`: uno por transacción. Por eso no hay phantoms; sí write skew.

VACUUM y un snapshot abierto

```
node mvcc.mjs vacuum

A: UPDATE shipment SET price = price WHERE shipment_id BETWEEN 1 AND 100
   UPDATE 100
A: VACUUM (VERBOSE) shipment
   tuples: 100 removed, 11982 remain, 0 are dead but not yet removable
   removable cutoff: 25433, which was 0 XIDs old when operation ended
B: BEGIN ISOLATION LEVEL REPEATABLE READ
B: SELECT count(*) FROM shipment
A: UPDATE shipment SET price = price WHERE shipment_id BETWEEN 1 AND 100
A: VACUUM (VERBOSE) shipment
   tuples: 0 removed, 11910 remain, 100 are dead but not yet removable
   removable cutoff: 25369, which was 65 XIDs old when operation ended
B: COMMIT
A: VACUUM (VERBOSE) shipment
   tuples: 100 removed, 11643 remain, 0 are dead but not yet removable
   removable cutoff: 25434, which was 0 XIDs old when operation ended
```

- Tupla muerta: una versión que ningún snapshot puede ver. `VACUUM` la quita.
- Con B abierta, el corte queda en el `xmin` de su snapshot: nada se quita.

Transacciones largas y wraparound

Una transacción abierta durante horas

Impide limpiar todo lo que se borró desde que empezó, en toda la base. Las tablas se inflan (bloat) y el espacio no vuelve solo. Causa típica: conexiones “idle in transaction”.

- Los xid son de 32 bits y se comparan en círculo: unos 2 000 millones de “pasado” y otros tantos de “futuro”.
- Sin cuidado, al dar la vuelta (wraparound) una tupla vieja quedaría “en el futuro”.
- `VACUUM` congela (freeze) las tuplas viejas: visibles para todos.
- Autovacuum lo fuerza pasadas `autovacuum_freeze_max_age` transacciones (200 millones).

SECCIÓN 07

Write skew

07



El viaje 12, en REPEATABLE READ

Vehículo de 60 kg. A asigna el envío 55 (48,20 kg); B, el 444 (43,50 kg). Cada uno revisa la carga y, si cabe, asigna.

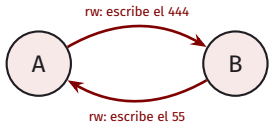
```
node skew.mjs rr

A: BEGIN ISOLATION LEVEL REPEATABLE READ
B: BEGIN ISOLATION LEVEL REPEATABLE READ
A: carga del viaje 12
   capacity_kg=60 load_kg=0 -> 48.2 kg cabe
B: carga del viaje 12
   capacity_kg=60 load_kg=0 -> 43.5 kg cabe
A: UPDATE shipment SET trip_id = 12 WHERE shipment_id = 55
B: UPDATE shipment SET trip_id = 12 WHERE shipment_id = 444
A: COMMIT
B: COMMIT

 capacity_kg | load_kg
-------------+---------
          60 |   91.70
(1 row)
```

Por qué no es serializable

- Filas distintas: SI no ve conflicto de escritura.
- Antidependencia Ti $\rightarrow$ Tj: Ti leyó algo y Tj escribió una versión nueva que Ti no vio.
- A leyó “envíos del viaje 12”; B escribió el 444 en el viaje 12: A $\rightarrow$ B.
- Y al revés: B $\rightarrow$ A. Ciclo.
- Si A va primero, B no asigna; si B va primero, A no asigna.



Write skew

Dos transacciones leen datos que se solapan, deciden con lo que leyeron y escriben partes distintas. Toda regla entre varias filas está expuesta.

Arreglo 1: `SERIALIZABLE` (SSI)

```
node skew.mjs serializable

A: UPDATE shipment SET trip_id = 12 WHERE shipment_id = 55
B: UPDATE shipment SET trip_id = 12 WHERE shipment_id = 444
A: COMMIT
B: COMMIT
   ERROR:  could not serialize access due to read/write dependencies among transactions  (código 40001)

 capacity_kg | load_kg
-------------+---------
          60 |   48.20
(1 row)
```

- SI + registro de lo leído. Todo ciclo de SI tiene dos antidependencias seguidas entre concurrentes: la estructura peligrosa. Se aborta una.
- Falló B, y en el `COMMIT`: antes no se sabía si se iba a completar.

Los `SIReadLock`

```
node skew.mjs serializable
```

```
 session | locktype |   relation    | page | tuple |    mode
---------+----------+---------------+------+-------+------------
 A       | relation | vehicle       |      |       | SIReadLock
 A       | tuple    | trip          |    0 |    12 | SIReadLock
 A       | page     | trip_pkey     |    1 |       | SIReadLock
 A       | relation | shipment      |      |       | SIReadLock
 A       | page     | shipment_pkey |    1 |       | SIReadLock
```

(las filas de A; las de B son iguales)

- No bloquean a nadie: marcan lo leído. Tupla, página o tabla.
- `shipment` entera: la suma hizo un sequential scan (no hay índice en `trip_id`).
- Por eso SSI aborta de más (falsos positivos): hay que reintentar siempre.

Arreglo 2: materializar el conflicto

```
node skew.mjs rc-lock

A: BEGIN ISOLATION LEVEL READ COMMITTED
B: BEGIN ISOLATION LEVEL READ COMMITTED
A: SELECT trip_id FROM trip WHERE trip_id = 12 FOR UPDATE
A: carga del viaje 12
   capacity_kg=60 load_kg=0 -> 48.2 kg cabe
B: SELECT trip_id FROM trip WHERE trip_id = 12 FOR UPDATE   -- espera
A: UPDATE shipment SET trip_id = 12 WHERE shipment_id = 55
A: COMMIT
   -- B continúa
B: carga del viaje 12
   capacity_kg=60 load_kg=48.20 -> 43.5 kg no cabe
B: COMMIT
```

La fila del viaje hace de lock del predicado “envíos del viaje 12”.

Lo mismo en REPEATABLE READ

```
node skew.mjs rr-lock

B: SELECT trip_id FROM trip WHERE trip_id = 12 FOR UPDATE   -- espera
A: UPDATE shipment SET trip_id = 12 WHERE shipment_id = 55
A: COMMIT
   -- B continúa
B: carga del viaje 12
   capacity_kg=60 load_kg=0 -> 43.5 kg cabe
B: UPDATE shipment SET trip_id = 12 WHERE shipment_id = 444
B: COMMIT
```

Pregunta

B también esperó el lock del viaje. ¿Por qué ahora no vio los 48,20 kg? ¿Cuándo tomó su snapshot?

- Arreglo 3: un trigger que revise la capacidad. Cuidado: lee con el mismo snapshot.

SECCIÓN 08

Timestamps y validación (opcional)

08

Timestamp ordering (sec. 18.5)

- TS(Ti) al empezar; cada Q guarda W-timestamp(Q) y R-timestamp(Q).
- `read(Q)`: si TS(Ti) $<$ W-ts(Q), rollback. Si no, lee y sube R-ts(Q).
- `write(Q)`: si TS(Ti) $<$ R-ts(Q) o TS(Ti) $<$ W-ts(Q), rollback. Si no, escribe y W-ts(Q) = TS(Ti).
- Rollback y reinicio con un timestamp nuevo.

- Conflict serializable; sin deadlocks (nadie espera).
- Starvation posible; puede no ser recuperable.

Regla de Thomas

Si TS(Ti) $<$ W-ts(Q) en un `write`, se ignora la escritura. Acepta schedules view serializables como `r1(A) w2(A) w1(A) w3(A)`.

Validación: control optimista (sec. 18.6)

1. **Lectura**: trabaja en copias locales.
2. **Validación**: ¿alguien escribió lo que leí?
3. **Escritura**: si pasó, aplica sus cambios; si no, rollback.

- TS(Ti) = Validation(Ti). Para toda Tk anterior: Finish(Tk) $<$ Start(Ti), o escrituras de Tk $\cap$ lecturas de Ti $= \emptyset$ y Finish(Tk) $<$ Validation(Ti).
- Conviene cuando los conflictos son raros.

En una aplicación

Una columna de versión: `UPDATE ... WHERE id = $1 AND version = $2`. Si no afectó filas, alguien la cambió: reintentar.

SECCIÓN 09

Cierre

09

Resumen

- 2PL: crecimiento y contracción; serializable por lock points. Strict: X hasta el final. Rigorous: todo, orden de commits.
- Lock manager: hash table y cola FIFO por ítem.
- Deadlocks: orden fijo, wait-die, wound-wait, timeouts, waits-for graph.
- PostgreSQL: `deadlock_timeout`, `40P01`; la espera por una fila es un `ShareLock` sobre un xid.
- IS, IX, S, SIX, X; `ROW EXCLUSIVE` $\approx$ IX.
- Phantom: conflicto sobre un predicado.
- MVCC: los lectores no esperan. SI: snapshot y first updater wins.
- `xmin`, `xmax`, `ctid`; snapshot `xmin:xmax:xip`; RC por sentencia, RR por transacción.
- `VACUUM` quita lo que ningún snapshot ve; congela por el wraparound.
- Write skew: SI lo deja pasar. `SERIALIZABLE` (SSI) o `FOR UPDATE` sobre una fila que represente la regla.

Para la próxima clase

- Laboratorio: locks y MVCC con tres terminales de `psql` (`code/week-14/locks.sql` para observar).
- Se entrega P4. Se publica **P5** (`quipu`: lock manager y recuperación); se entrega en la semana 16.
- Leer Silberschatz, secs. 19.1 a 19.5 (recuperación y WAL) y la nota 14 de CS186.
- Correr `make test` en `code/week-14` y leer los apuntes de la semana.

Pregunta

Un `ROLLBACK` en PostgreSQL deja `xmax` escrito en las tuplas y no borra las versiones nuevas. ¿Qué tiene que guardar el motor para que, después de una caída, nadie vea lo que hizo una transacción que no llegó al commit?