

UNIVERSIDAD NACIONAL DE INGENIERÍA

Bases de Datos

Recuperación: log, WAL y ARIES

Transacciones y sistemas distribuidos

Gerson Garrido

Semana 15

Hoy

Sesión A: fallos, log y WAL

1. Fallos y almacenamiento estable
2. El log: registros, undo y redo
3. Steal y force: qué exige cada uno
4. La regla WAL y el algoritmo del libro
5. El WAL de PostgreSQL por dentro

Sesión B: ARIES y checkpoints

6. LSN, pageLSN, CLR, TT y DPT
7. Fuzzy checkpoints
8. Analysis, redo y undo: un ejemplo completo
9. Checkpoints y una caída en PostgreSQL
10. Backups, PITR y servidores de respaldo

Lectura

Silberschatz, secs. 19.1–19.5 (sesión A) y 19.9, 19.6–19.7 (sesión B). CS186, *Recovery*: de *Force/No Force* a *WAL Implementation*, *Undo Logging* y *Redo Logging* (A); de *Aborting a Transaction* a *Undo Example* (B). Además: CMU 15-445 (Fall 2025), *Database Logging* y *Database Recovery*; Mohan et al., *ARIES* (TODS 1992); PostgreSQL, *Reliability*, *Write-Ahead Logging (WAL)*, *WAL Configuration* y *Continuous Archiving and Point-in-Time Recovery*.

SECCIÓN 01

Lo que se pierde en una caída

01

La A y la D de ACID

- Semana 14: el aislamiento, con locks y MVCC.
- Quedan dos letras, y las dos son del **sistema de recuperación**:
  - **Atomicidad**: lo que no terminó no deja rastro.
  - **Durabilidad**: lo que hizo commit sobrevive a cualquier caída posterior.
- Semana 9: las páginas sucias viven en el buffer pool y se escriben cuando le conviene al buffer manager.

El problema

En el momento de una caída puede haber en disco cambios de transacciones sin commit, y pueden faltar cambios de transacciones con commit.

Un envío a medias

entrega.sql

```sql
BEGIN;
INSERT INTO shipment_event
  (shipment_id, seq, at, agency_id, kind)
VALUES (503, 2, now(), 20, 'delivered');
UPDATE shipment
SET status = 'delivered', delivered_at = now()
WHERE shipment_id = 503;
COMMIT;
```

Un corte de luz…

1. …entre el `INSERT` y el `UPDATE`, con la página del evento ya en disco: hay que deshacer.
2. …justo después del `COMMIT`, sin ninguna página en disco: hay que rehacer.
3. …a la mitad de la escritura de una página de 8 KiB: una **torn page**, ni la versión vieja ni la nueva.

La herramienta

El **log**: antes de tocar los datos, se anota qué se va a cambiar (al final de un archivo: secuencial y barato).

SECCIÓN 02

Fallos y almacenamiento

02

Clasificación de fallos

Fallo	Ejemplo	Qué se pierde
De una transacción: error lógico	`weight_kg = 0` viola un `CHECK`	nada: rollback de esa transacción
De una transacción: error del sistema	víctima de un deadlock (`40P01`)	nada: rollback, se puede reintentar
Caída del sistema	corte de luz, bug, kernel panic	la memoria volátil; el disco queda intacto (**fail-stop**)
Fallo del disco	sector ilegible, SSD gastado, `rm` equivocado	almacenamiento no volátil: hacen falta copias afuera

Error común

Un RAID protege contra la pérdida de un disco, no contra un `DELETE` equivocado: el `DELETE` también se copia en todos los discos.

Volátil, no volátil y estable

Tipo	Sobrevive a
Volátil (RAM)	nada
No volátil (disco, SSD)	caídas, no fallos del propio dispositivo
Estable	se supone que a todo

- El estable es una idealización: varias copias con fallos independientes.
- El log se escribe en almacenamiento estable.

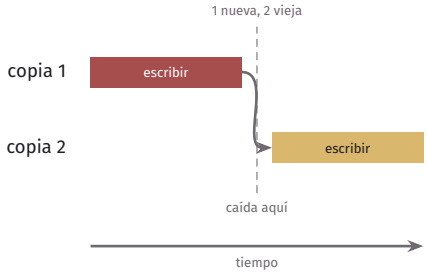
El modelo del libro

- `input(B)`, `output(B)`: entre disco y buffer.
- `read(X)`, `write(X)`: entre el buffer y la copia privada de la transacción.

Lo importante

`write(X)` no escribe en disco. El `output` lo decide el buffer manager, antes o después del commit.

Almacenamiento estable con dos copias



1. Escribir la copia 1.
2. Cuando terminó bien, escribir la copia 2.
3. Completa solo cuando terminó la 2.

Al recuperar, con checksums:

- Una dañada: se reemplaza por la otra.
- Las dos bien pero distintas: se copia una sobre la otra (completa o sin efecto).

Por qué una después de la otra

Si se escribieran a la vez, un corte podría dañar las dos.

SECCIÓN 03

El log

03

Registros del log

Registro	Significado
$\langle T_i \text{ start} \rangle$	T_i empezó
$\langle T_i, X_j, V_1, V_2 \rangle$	X_j tenía V_1 (undo) y ahora V_2 (redo)
$\langle T_i \text{ commit} \rangle$	T_i hizo commit
$\langle T_i \text{ abort} \rangle$	T_i terminó su rollback

Commit

T_i **hizo commit** cuando $\langle T_i \text{ commit} \rangle$ llegó a almacenamiento estable. Un instante antes, no.

- **Modificación diferida**: nada toca la base hasta el commit. Nunca undo.
- **Modificación inmediata**: los buffers cambian mientras corre. Es lo que hacen los motores.

Undo y redo con el log

- **redo(T_i)**: hacia adelante, cada ítem toma su valor nuevo.
- **undo(T_i)**: hacia atrás, cada ítem toma su valor viejo, y se anota $\langle T_i, X_j, V \rangle$ (redo-only); al final, $\langle T_i$ abort$\rangle$.

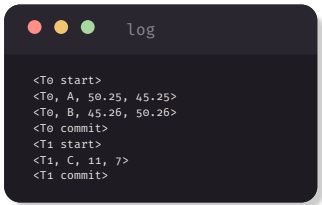
Después de una caída (sec. 19.3)

- `start` sin `commit` ni `abort` $\Rightarrow$ **undo(T_i)**.
- `start` y `commit` o `abort` $\Rightarrow$ **redo(T_i)**.

¿Rehacer una transacción que abortó?

Su rollback dejó en el log los registros redo-only. Rehacer todo, en orden, deja los valores viejos. Es la primera aparición de **repetir la historia**.

Un ejemplo pequeño



A, B: `price` de 501 y 502. C: `vehicle_id` del viaje 12.

El log llega hasta	Decisión	Resultado
$\langle T_0, B, \ldots \rangle$	undo(T_0)	A = 50.25, B = 45.26
$\langle T_1, C, \ldots \rangle$	redo(T_0), undo(T_1)	A = 45.25, B = 50.26, C = 11
$\langle T_1$ commit$\rangle$	redo de las dos	A = 45.25, B = 50.26, C = 7

Qué se guarda en V_1 y V_2

Physical: bytes de la página. **Logical**: la operación. **Physiological**: física para elegir la página, lógica adentro (“página 147, slot 65, esta tupla”). Los motores usan physiological.

SECCIÓN 04

Steal y force

04

Dos decisiones del buffer manager

Force / no-force

Force: en el commit, se escriben en disco todas las páginas de la transacción.
No-force: el commit solo escribe el log.

Steal / no-steal

Steal: se puede escribir en disco una página con cambios sin commit.
No-steal: esa página no sale hasta que la transacción termine.

- ¿Hace falta **undo**? Solo si en disco puede haber cambios sin commit: steal.
- ¿Hace falta **redo**? Solo si un commit puede no tener sus páginas en disco: no-force.

Las cuatro combinaciones

	force	no-force
no-steal	ni undo ni redo	solo redo
steal	solo undo	**undo y redo**

lo que eligen los motores

La esquina sin recuperación cuesta mucho:

- **No-steal**: todo lo que cambia una transacción tiene que caber en memoria. Un `UPDATE` de todo `shipment` en `chasqui_big`: 24 667 páginas, 16 384 marcos.
- **Force**: una escritura aleatoria por página en cada commit, y la misma página caliente se escribe una y otra vez.

Steal + no-force

La más rápida y la que más le exige al log: “lo que no hizo commit puede estar en disco” (undo) y “lo que hizo commit puede no estar” (redo).

Undo logging y redo logging (CS186)

Undo logging: steal + force

- El update guarda el valor **viejo**.
- Log del cambio antes que la página.
- Todas las páginas antes que el commit.
- Al recuperar: hacia atrás, restaurar lo que no terminó.

Redo logging: no-steal + no-force

- El update guarda el valor **nuevo**.
- Update y commit en el log antes que cualquier página.
- Al recuperar: hacia adelante, rehacer las que hicieron commit.

Por eso dos valores

Cada esquema paga con una política cara. Guardar V_1 y V_2 es lo que permite steal y no-force a la vez.

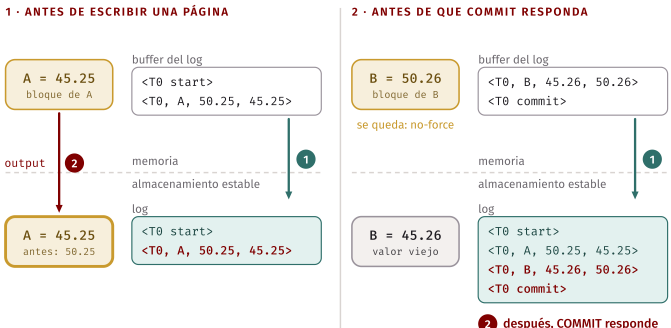
SECCIÓN 05

La regla WAL

05



Write-ahead logging: las dos partes



Error común

`COMMIT` no escribe las páginas de datos. Solo fuerza el log hasta el registro de commit; las páginas las escriben después el background writer y el checkpointer.

El buffer del log, group commit y `fsync`

- El log también tiene un **buffer en memoria**; se escribe por bloques.
- Es obligatorio forzarlo antes de escribir una página (hasta su último registro) y en cada commit.
- **Group commit**: varios commits casi al mismo tiempo $\Rightarrow$ una sola escritura del log para todos.
- `write` deja los datos en la caché del sistema operativo. Hace falta `fsync`. Y el disco puede mentir con su propia caché sin batería.

Con Sistemas Operativos

Un file system con journaling (ext4) anota la operación sobre los metadatos antes de hacerla y la rehace al montar. Es la idea del WAL; pero el log de un DBMS conoce las transacciones y puede deshacer las que no terminaron.

SECCIÓN 06

El algoritmo del libro

06

Checkpoints y rollback normal

Checkpoint simple (sec. 19.3)

Sin updates mientras tanto:

1. escribir el log que está en memoria;
2. escribir todas las páginas sucias;
3. escribir $\langle \text{checkpoint } L \rangle$, con L = transacciones activas.

Acota cuánto log hay que leer, pero detiene el sistema.

Rollback normal de T_i (sec. 19.4)

1. Recorrer el log hacia atrás.
2. Por cada $\langle T_i, X_j, V_1, V_2 \rangle$: poner V_1 y anotar $\langle T_i, X_j, V_1 \rangle$.
3. En $\langle T_i \text{ start} \rangle$: anotar $\langle T_i \text{ abort} \rangle$.

Un rollback también escribe en el log.

Recuperación después de una caída (sec. 19.4)

Fase de redo: hacia adelante

Desde el último checkpoint, con undo-list $= L$:

- rehacer todos los updates y los redo-only;
- `start`: entra a la undo-list;
- `commit` o `abort`: sale.

Fase de undo: hacia atrás

Desde el final:

- deshacer los updates de la undo-list, anotando redo-only;
- en su `start`: anotar `abort` y sacarla;
- terminar cuando la lista está vacía (puede cruzar el checkpoint).

Idempotencia

Si cae durante la recuperación, se vuelve a empezar. Escribir “A = 45.25” es idempotente; “restar 5 a A”, no. Por eso el log guarda valores.

Un ejemplo con checkpoint y rollback

log

```
 1  <T1 start>
 2  <T1, A, 50.25, 45.25>
 3  <T2 start>
 4  <checkpoint {T1, T2}>
 5  <T2, B, 45.26, 40.00>
 6  <T1 commit>
 7  <T3 start>
 8  <T3, C, 114.90, 120.00>
 9  <T2, A, 45.25, 47.00>
10  <T2, A, 45.25>
11  <T2, B, 45.26>
12  <T2 abort>
13  <T3, A, 45.25, 44.00>
---- caída ----
```

Redo desde 4

	acción	undo-list
5	B = 40.00	{T1, T2}
6	commit	{T2}
7	start	{T2, T3}
8	C = 120.00	
9	A = 47.00	
10	A = 45.25	
11	B = 45.26	
12	abort	{T3}
13	A = 44.00	{T3}

Undo de T3

	acción	se anota
13	A = 45.25	⟨T3, A, 45.25⟩
8	C = 114.90	⟨T3, C, 114.90⟩
7	start	⟨T3 abort⟩

Resultado: A = 45.25 (T1), B = 45.26, C = 114.90. El redo de 9 y 10 repite el rollback de T2 sin preguntar nada.

SECCIÓN 07

El WAL de PostgreSQL

07



PostgreSQL: steal, no-force y sin undo

- El buffer pool es steal y no-force. Necesitaría undo y redo…
- …pero el WAL es solo de redo.

Por qué: MVCC

Un `UPDATE` escribe una versión nueva (`xmin`) y marca la vieja (`xmax`). La visibilidad se decide con `pg_xact`: si la transacción no hizo commit, su versión es invisible y la vieja sigue visible. Deshacer es no hacer nada; `VACUUM` limpia después.

- Oracle e InnoDB guardan las versiones viejas aparte (undo log, rollback segments): esos sí tienen fase de undo.
- El WAL vive en `pg_wal`, en segmentos de 16 MB.

El LSN es una posición en bytes

wal.sql

```
        name        | setting  | unit
--------------------+----------+------
 checkpoint_timeout | 300      | s
 data_checksums     | on       |
 full_page_writes   | on       |
 max_wal_size       | 1024     | MB
 synchronous_commit | on       |
 wal_level          | replica  |
 wal_segment_size   | 16777216 | B
(7 rows)

    lsn     |         segment
------------+-------------------------
 1/A6E7FE70 | 000000010000000100000A6
(1 row)
```

- `1/A6E7FE70` = 0x1A6E7FE70: 64 bits, en dos mitades.
- Dividido por 16 MB (2^{24}): segmento 0x1A6.
- El nombre: timeline `00000001`, y el segmento en dos partes.
- El WAL solo crece: los LSN crecen.

Los registros de una transacción

wal.sql

```sql
CHECKPOINT;
SELECT pg_current_wal_insert_lsn() AS antes \gset
BEGIN;
INSERT INTO shipment_event (shipment_id, seq, at, agency_id, kind)
SELECT shipment_id, seq + 1, now(), agency_id, kind
FROM shipment_event WHERE shipment_id = 501 ORDER BY seq DESC LIMIT 1;
UPDATE shipment SET price = price + 1 WHERE shipment_id = 501;
COMMIT;
SELECT pg_current_wal_insert_lsn() AS despues \gset
SELECT ... FROM pg_get_wal_records_info(:'antes', :'despues') WHERE ...;
```

- `pg_walinspect` lee el WAL registro por registro.
- `block_ref` dice qué página toca cada registro; `pg_filenode_relation` la traduce a un nombre.
- El `WHERE` deja solo esta transacción y esta base: el servidor es compartido.

Dos sentencias, trece registros

chasqui_w15, otra corrida

```
  xid  |    rmgr     |   record_type    | len  | fpi  |         rel          | blk
-------+-------------+------------------+------+------+----------------------+-----
     0 | XLOG        | FPI_FOR_HINT     | 8157 | 8108 | pg_statistic         |  22
     0 | XLOG        | FPI_FOR_HINT     | 8241 | 8192 | pg_statistic         |  23
 28917 | Heap        | INSERT           | 2838 | 2784 | shipment_event       | 341
 28917 | Btree       | INSERT_LEAF      | 7473 | 7420 | shipment_event_pkey  |   7
 28917 | XLOG        | FPI_FOR_HINT     | 8221 | 8172 | pg_statistic         |  20
 28917 | Heap        | LOCK             | 8167 | 8108 | shipment             |   6
 28917 | Heap        | LOCK             | 2111 | 2052 | agency               |   0
 28917 | XLOG        | FPI_FOR_HINT     | 7877 | 7828 | pg_proc              |  33
 28917 | Heap2       | PRUNE_ON_ACCESS  |   52 |    0 | pg_proc              |  33
 28917 | Heap        | LOCK             |   54 |    0 | shipment             |   6
 28917 | Heap        | UPDATE           | 6489 | 6416 | shipment             | 147
 28917 | Btree       | INSERT_LEAF      | 7453 | 7400 | shipment_pkey        |   2
 28917 | Transaction | COMMIT           |   34 |    0 |                      |
(13 rows)
```

Qué es cada registro

- **`rmgr`** (resource manager): qué parte del motor sabe aplicarlo.
- `Heap INSERT` y `Btree INSERT_LEAF`: la tupla y su entrada en el índice de la clave primaria. Cada índice tiene sus propios registros.
- Dos `Heap LOCK` (`shipment` 6, `agency` 0): las claves foráneas. La fila referenciada se bloquea `FOR KEY SHARE`, y el lock vive en la tupla.
- `LOCK` + `UPDATE` en `shipment`: la página 6 está llena; la versión nueva va a la 147. No es HOT ⇒ otra entrada en `shipment_pkey`.
- `Transaction COMMIT`: el $\langle T_i$ commit$\rangle$ del libro, 34 bytes.
- `FPI_FOR_HINT`, `PRUNE_ON_ACCESS`: el catálogo, leído al planear. Con data checksums, cambiar hint bits también se anota.

`prev_lsn` no es el prevLSN de ARIES

En PostgreSQL apunta al registro anterior de **todo** el WAL, de cualquier transacción. Sin undo, no hace falta encadenar los registros de cada transacción.

Full-page writes

- El `INSERT` mide 2838 bytes: 2784 son la **página entera** (FPI).
- Contra las torn pages: la primera modificación de una página después de un checkpoint lleva su imagen. El redo empieza en el checkpoint y la copia encima.
- Sin el hueco libre: la imagen de `agency` mide $8192 - 6140 = 2052$ (semana 9).

El mismo `UPDATE`, sin checkpoint antes:

chasqui_w15, otra corrida

```
 start_lsn  |  end_lsn   |  xid  |   record_type    | len | fpi | blk
------------+------------+-------+------------------+-----+-----+-----
 2/7485C240 | 2/7485C278 |     0 | PRUNE_ON_ACCESS  |  56 |   0 |   6
 2/7485C278 | 2/7485C2C0 | 28918 | HOT_UPDATE       |  71 |   0 | 147
 2/7485C2C0 | 2/7485C2E8 | 28918 | COMMIT           |  34 |   0 |
(3 rows)
```

71 bytes y sin imagen: la página 147 ya tuvo la suya. Justo después de un checkpoint, el WAL crece mucho más rápido.

El pageLSN de una página real

chasqui_w15, otra corrida

```
   ctid
----------
 (147,65)
(1 row)

  page_lsn
------------
 2/7485C2C0
(1 row)
```

- `pd_lsn`, los primeros 8 bytes del encabezado (semana 9).
- Es el `end_lsn` del `HOT_UPDATE`: el último registro que cambió la página.
- PostgreSQL guarda el final del registro; ARIES, el comienzo. La idea es la misma.
- Antes de escribir la página, el buffer manager fuerza el WAL hasta su `pd_lsn`: la parte (1) de la regla WAL.

Commit síncrono y asíncrono

`synchronous_commit = off`

El commit responde con el registro en el buffer del WAL; el WAL writer lo escribe cada `wal_writer_delay` (200 ms). Una caída puede perder los últimos commits (unas tres veces ese tiempo), pero nunca deja la base inconsistente.

`fsync = off`

PostgreSQL deja de saber en qué orden llegan las cosas al disco. Una página puede llegar antes que su WAL: se rompe la parte (1) y la base puede quedar **corrupta**.

- Group commit: el primero que hace `fsync` arrastra los commits que ya estaban en el buffer. `commit_delay` agrega una espera a propósito.

SECCIÓN 08

ARIES

08

ARIES: tres ideas

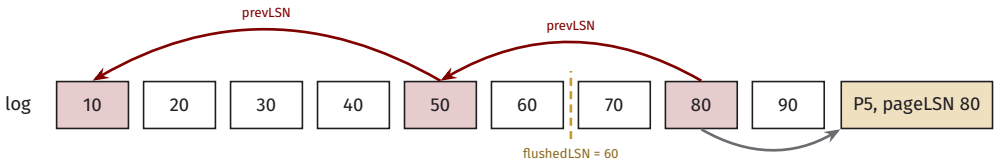
- Mohan et al., IBM, 1992. Base de DB2, SQL Server y muchos otros. Steal + no-force.

1. **Write-ahead logging**.
2. **Repetir la historia** en el redo: dejar la base como estaba en la caída, con los cambios de todas las transacciones. Recién después, undo.
3. **Anotar lo que hace el undo** (CLR), para no deshacer dos veces, aunque haya caídas repetidas.

Lo que agrega

Números que permiten saltarse trabajo: comparar dos LSN para saber, sin leer la página, que no hay nada que rehacer en ella.

LSN, pageLSN, flushedLSN y prevLSN



- **LSN**: número único y creciente de cada registro.
- **prevLSN**: el registro anterior de la misma transacción (T1: 80 $\rightarrow$ 50 $\rightarrow$ 10).
- **flushedLSN**: hasta dónde está el log en almacenamiento estable.
- **pageLSN**: el último registro que cambió la página. Llega a disco con ella.

La regla WAL como comparación

Una página solo puede ir a disco si **pageLSN** $\leq$ **flushedLSN**. Aquí no: primero hay que forzar el log hasta 80.

Dos tablas en memoria

Transaction table (TT)

Por cada transacción activa: estado (running, committed, aborting) y **lastLSN**, el comienzo de su cadena de prevLSN.

Dirty page table (DPT)

Por cada página sucia: **recLSN**, el primer registro que la ensució desde que se escribió. Todo lo anterior ya está en disco. Al escribirse, sale.

- En la DPT de memoria: recLSN $>$ pageLSN en disco (el cambio que la ensució no llegó). En la que reconstruye la recuperación, no siempre: escribir una página no se anota.
- Las dos tablas se pierden en la caída. Hay que reconstruirlas.

Compensation log records

CLR

Se escribe al deshacer un update. Describe el cambio que restauró el valor viejo, y lleva **undoNextLSN**: el siguiente registro de esa transacción que falta deshacer (el prevLSN del update deshecho).

- Un CLR se rehace.
- Un CLR nunca se deshace: el undo salta a su undoNextLSN.

Rollback y commit

- Rollback: `abort`, un CLR por update siguiendo los prevLSN, `end`.
- Commit: `commit`, forzar el log, responder, `end`.
- `end`: ya no necesita nada del log.

100 updates, cae después de deshacer 60: el siguiente arranque salta directo a los 40 que faltan.

Fuzzy checkpoints

1. `begin_checkpoint`.
2. `end_checkpoint` con una copia de la **TT** y de la **DPT**. Las transacciones siguen trabajando.
3. Cuando está en almacenamiento estable: el **registro maestro**, en un lugar fijo del disco, apunta al `begin_checkpoint`.

- No detiene nada y no escribe páginas. La DPT ya dice desde dónde hay que rehacer.
- El buffer manager escribe páginas en segundo plano: los recLSN avanzan.
- El libro lo presenta como un solo registro; CS186, como el par begin/end.

Error común

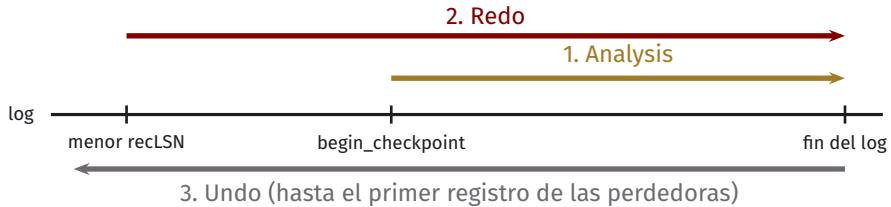
Un checkpoint no hace commit de nada y no vuelve inútil el log anterior: en el ejemplo que viene, el redo y el undo llegan antes del checkpoint. Y un PITR necesita todo el log archivado.

SECCIÓN 09

Las tres fases

09

Las tres pasadas



- **Analysis**: reconstruye la TT y la DPT; da las perdedoras y el menor recLSN.
- **Redo**: repite la historia desde el menor recLSN, que puede ser anterior al checkpoint.
- **Undo**: hacia atrás, solo las perdedoras.

Error común

Creer que la recuperación solo rehace lo que hizo commit. ARIES rehace **todo** y después deshace a las perdedoras.

Analysis

Desde el `begin_checkpoint`, hacia adelante; en el `end_checkpoint` se combinan sus tablas con lo ya visto (TT: lo visto es más reciente; DPT: el menor recLSN). Con cada registro:

- si no está en la TT, entra (running); su **lastLSN** = este LSN;
- `commit` → committed; `abort` → aborting; `end` → sale;
- update o CLR: si la página no está en la DPT, entra con recLSN = este LSN. Si estaba, no cambia.

Al final

- **Perdedoras**: las de la TT que no hicieron commit.
- El redo empieza en el **menor recLSN** de la DPT.

La DPT es conservadora: puede tener páginas que ya se escribieron (escribir una página no se anota). El redo lo va a descubrir.

Redo y undo

Redo

Desde el menor recLSN, cada update y CLR se rehace, **salvo** que:

1. la página no esté en la DPT;
2. su LSN sea menor que el recLSN de la página;
3. el pageLSN en disco sea $\geq$ su LSN.

(1) y (2) sin leer la página. Rehacer: aplicar y poner pageLSN = LSN. No se anota nada.

Undo

Conjunto con el lastLSN de cada perdedora. Tomar siempre el mayor:

- update: deshacer, CLR con undoNextLSN = prevLSN, seguir con el prevLSN;
- CLR: saltar a su undoNextLSN;
- abort: seguir con su prevLSN;
- sin siguiente: escribir `end`.

SECCIÓN 10

Un ejemplo completo

10

El log en el momento de la caída

example.mjs

```
LSN  registro                    prevLSN
10   T1 update P5                -       shipment 501.status: in_transit → delivered
20   T2 update P3                -       shipment 233.price: 30 → 35
30   begin_checkpoint
40   end_checkpoint  TT = {T1: 10, T2: 20}  DPT = {P5: 10}
50   T1 update P3                10      shipment 240.price: 18.5 → 20
60   T2 update P7                20      trip 12.vehicle_id: 11 → 7
70   T2 commit                   60
80   T1 update P5                50      shipment 502.status: registered → in_transit
90   T3 update P1                -       trip 40.vehicle_id: 1 → 3
100  T2 end
---- caída ----
En disco: pageLSN(P1) = 0, pageLSN(P3) = 50, pageLSN(P5) = 0, pageLSN(P7) = 0
```

- P3 no está en la DPT del checkpoint: se escribió entre 20 y 30.
- P3 tiene pageLSN 50 en disco: un cambio de T1, sin commit. Steal.
- T2 hizo commit, pero P7 tiene pageLSN 0. No-force.

Analysis desde 30

LSN	efecto
40	TT = {T1: 10, T2: 20}, DPT = {P5: 10}
50	lastLSN(T1) = 50; P3 entra con 50
60	lastLSN(T2) = 60; P7 entra con 60
70	T2 committed
80	lastLSN(T1) = 80; P5 ya estaba (10)
90	T3 entra; P1 entra con 90
100	T2 sale

Resultado

- TT = {T1: 80, T3: 90}
- Perdedoras: **T1 y T3**
- DPT = {P5: 10, P3: 50, P7: 60, P1: 90}
- El redo empieza en **10**

10 es anterior al checkpoint: P5 está sucia desde entonces.

Redo desde 10

LSN	registro	decisión
10	T1 update P5	en la DPT, $10 \geq 10$, pageLSN $0 < 10$: **se rehace**; pageLSN(P5) = 10
20	T2 update P3	$20 <$ recLSN 50: se salta sin leer la página
50	T1 update P3	$50 \geq 50$, se lee P3: pageLSN $50 \geq 50$, se salta
60	T2 update P7	**se rehace**: el cambio de T2, que hizo commit
80	T1 update P5	pageLSN(P5) es $10 < 80$: **se rehace**
90	T3 update P1	**se rehace**

Después del redo

Las cuatro páginas están como en el momento de la caída, con los cambios de T1 y T3. Commit, end y checkpoints no se rehacen.

Undo: conjunto {80, 90}

1. 90 (T3, P1): **CLR 110**, undoNextLSN –. **T3 end** en 120.
2. 80 (T1, P5): **CLR 130**, undoNextLSN 50.
3. 50 (T1, P3): **CLR 140**, undoNextLSN 10.
4. 10 (T1, P5): **CLR 150**, undoNextLSN –. **T1 end** en 160.

Dos convenciones

El ejemplo sigue al libro: el undo escribe CLR y `end`. CS186 además escribe un `abort` por perdedora al final de analysis: sus CLR quedan con otros LSN. Las páginas terminan igual.

Las tres fases, sobre el log



El conjunto queda vacío y la recuperación termina: en las páginas solo quedan los cambios de T2

El simulador: `aries.mjs`

- No es un motor: solo cuentas sobre un arreglo de registros.
- `analysis(log)`, `redo(log, a, disk)`, `undo(log, a, pages)`, `restart(log, disk)`.
- `node trace.mjs`: las tres fases del ejemplo.

example.mjs

```
{ lsn: 50, type: 'update',
  txn: 'T1', page: 'P3',
  item: 'shipment 240.price',
  before: 18.5, after: 20,
  prevLSN: 10 }
```

code/week-15

```
$ node trace.mjs
...
  TT  = {T1: running 80, T3: running 90}
  DPT = {P5: 10, P3: 50, P7: 60, P1: 90}
  perdedoras: T1, T3; el redo empieza en 10
Redo
    10  T1 update P5       se rehace
    20  T2 update P3       se salta (20 < recLSN 50)
    50  T1 update P3       se salta (pageLSN 50 ≥ 50)
...
Undo
    90  T3 update P1       CLR 110, undoNextLSN -; end en 120
    80  T1 update P5       CLR 130, undoNextLSN 50
    50  T1 update P3       CLR 140, undoNextLSN 10
    10  T1 update P5       CLR 150, undoNextLSN -; end en 160
...
Páginas al final
  P1 (pageLSN 110): trip 40.vehicle_id = 1
...
  P7 (pageLSN 60): trip 12.vehicle_id = 7
```

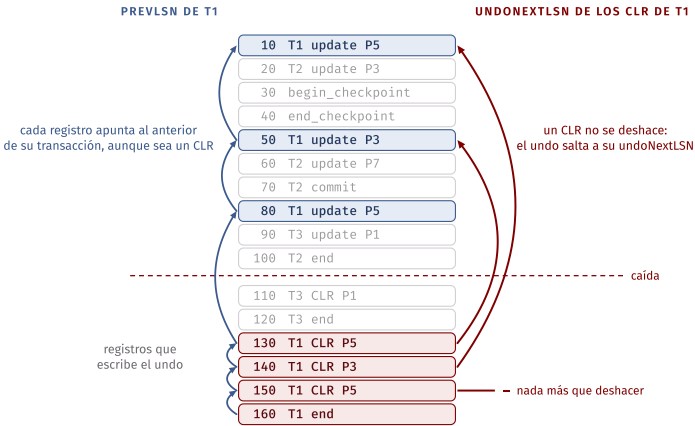
Una segunda caída, después del CLR 130

```
node trace.mjs 130

  110  T3 CLR P1          lastLSN(T3) = 110
  120  T3 end             T3 sale de la TT
  130  T1 CLR P5          lastLSN(T1) = 130
  TT  = {T1: running 130}
  DPT = {P5: 10, P3: 50, P7: 60, P1: 90}
  perdedoras: T1; el redo empieza en 10
Redo
...
  110  T3 CLR P1          se rehace
  130  T1 CLR P5          se rehace
Undo
  130  T1 CLR P5          se salta a undoNextLSN 50
   50  T1 update P3       CLR 140, undoNextLSN 10
   10  T1 update P5       CLR 150, undoNextLSN -; end en 160
```

- T3 ya no es perdedora: su `end` llegó.
- El redo rehace los CLR.
- El undo salta de 130 a 50: el cambio 80 no se deshace dos veces.
- El trabajo está acotado aunque se repitan las caídas.

Las dos cadenas de T1



Si el sistema cae otra vez justo después de 130, el undo empieza en el CLR 130 y salta a 50: el 80 no se deshace dos veces.

SECCIÓN 11

Checkpoints y caídas en PostgreSQL

11



El checkpoint de PostgreSQL

- No detiene a las transacciones, pero sí escribe todas las páginas que estaban sucias cuando empezó. Sin DPT.
- La recuperación empieza en el **punto de redo**, su comienzo.
- `CHECKPOINT_REDO` al empezar, `CHECKPOINT_ONLINE` al terminar. `pg_control`: el registro maestro.

wal.sql

```
  redo_lsn  | checkpoint_lsn
------------+----------------
 1/A6E7FF08 | 1/A6E7FF60
(1 row)
```

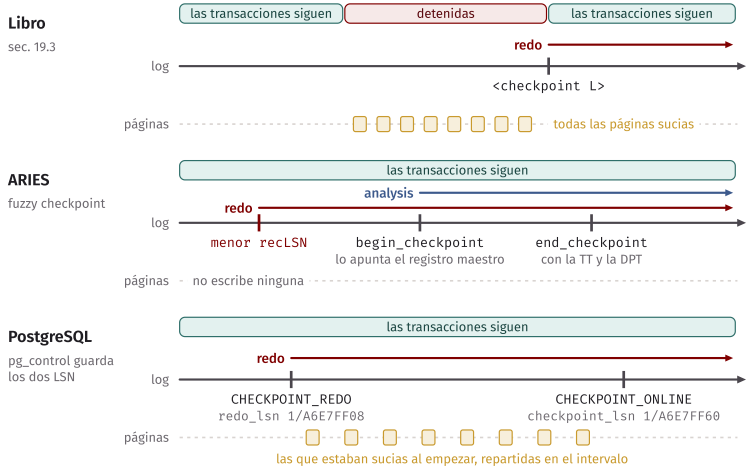
¿Cuándo?

- cada `checkpoint_timeout` (300 s);
- si el WAL creció `max_wal_size` (1 GB);
- con `CHECKPOINT`.

Se reparte en el tiempo: `checkpoint_completion_target = 0.9`.

Checkpoints seguidos: recuperaciones cortas, pero más escrituras y más full-page writes.

Tres checkpoints



Una caída de verdad: `make crash`

Cluster de prueba (`/tmp`, puerto 5415): 1000 filas con commit, un `UPDATE` sin commit, `pg_ctl -m immediate stop`.

crash.sh

```
redo_lsn del último checkpoint: 0/178F720
commit de 1000 filas; WAL insertado hasta 0/17B3EF0
transacción 754 abierta: UPDATE payment SET amount = 0 (sin commit)
WAL insertado hasta 0/17E9998
== Caída: pg_ctl -m immediate stop
== Arranque (líneas nuevas del log del servidor)
...
LOG:  database system was interrupted; last known up at 2026-11-01 20:29:04 UTC
LOG:  database system was not properly shut down; automatic recovery in progress
LOG:  redo starts at 0/178F720
LOG:  invalid magic number 0000 in WAL segment 000000010000000000000001, LSN 0/17E8000, offset 8290304
LOG:  redo done at 0/17E7F80 system usage: CPU: user: 0.00 s, system: 0.00 s, elapsed: 0.00 s
LOG:  checkpoint starting: end-of-recovery immediate wait
...
LOG:  database system is ready to accept connections
== Después de la recuperación
filas y total: 1000 10000.00
filas con xmax = 754: 969
estado de la transacción 754: aborted
registros del WAL con xid 754: 2922; de ellos COMMIT o ABORT: 0
```

Lo que dice esa salida

- `redo starts at` = el `redo_lsn` del último checkpoint.
- `invalid magic number`: ahí se acaba el WAL válido en disco (ceros). `redo done at`: el último registro rehecho.
- El final del WAL del `UPDATE` sin commit estaba en el buffer y se perdió. Ninguna página con esos cambios pudo llegar a disco sin él (regla WAL).
- Las 1000 filas están: su commit forzó el WAL. Durabilidad.
- El total es 10 000, no 0, pero 969 filas tienen `xmax = 754`: el redo repitió la historia de una transacción sin commit (el número cambia en cada corrida).
- `pg_xact_status`: `aborted`. No hay `ABORT` en el WAL: basta con que su commit no exista. No hay fase de undo.

Nunca

Borrar archivos de `pg_wal` para “liberar espacio”: sin ese WAL, la recuperación no puede rehacer lo que hizo commit.

SECCIÓN 12

Cuando se pierde el disco

12

Backups y point-in-time recovery (sec. 19.6)

- Una copia completa (dump), guardada fuera del servidor; puede ser fuzzy.
- **Todo el log** desde el comienzo de esa copia, también afuera.
- Restaurar la copia y rehacer el log: queda consistente, como después de una caída.
- Se puede detener el redo en cualquier punto: **point-in-time recovery**. `DELETE FROM shipment` a las 10:15 $\Rightarrow$ copia de la noche + log hasta las 10:14.

En PostgreSQL

`pg_basebackup` (copia física con el servidor funcionando) + archivado del WAL (`archive_mode`, `archive_command`). Al recuperar: `restore_command` y `recovery_target_time`. `pg_dump` es otra cosa: un dump lógico que no se combina con el WAL.

Servidores de respaldo (sec. 19.7)

- Otro servidor, en otro lugar, recibe el log del primario y lo aplica sin parar.
- Detectar la falla, pasar el control, tiempo hasta estar listo.

One-safe	commit con el log en el primario; se pueden perder los últimos
Two-very-safe	espera a los dos; sin respaldo, no hay commit
Two-safe	two-very-safe si los dos funcionan; si no, one-safe

En PostgreSQL

Streaming replication: el standby rehace el WAL con el mismo código que la recuperación. Asíncrona por defecto (one-safe). Con `synchronous_standby_names`, `synchronous_commit` elige: `remote_write`, `on`, `remote_apply`.

Semana 16: replicación y 2PC.

SECCIÓN 13

Cierre

13

El código de la semana

- `wal.sql`: sobre `chasqui_w15`, con `\gset`.
- `crash.sh`: `make crash`; el cluster se crea y se borra.
- `aries.mjs`, `example.mjs`, `trace.mjs` (`make trace`).
- Las pruebas no comparan LSN: cambian en cada corrida.
- El simulador no es P5: no tiene páginas ni buffer pool.

code/week-15

```
$ make test
...
✓ redo: 20 se salta por recLSN, 50 por pageLSN; los demás updates se rehacen
...
✓ segunda caída después del CLR 130: el cambio 80 no se deshace dos veces
...
✓ sobreviven las 1000 filas con commit; el UPDATE sin commit no aparece
...
✓ prev_lsn apunta al registro anterior del WAL, no al anterior de la transacción
ℹ tests 19
ℹ pass 19
ℹ fail 0
```

Resumen

- Fallos: de transacción, caída del sistema (fail-stop), disco. Almacenamiento volátil, no volátil y **estable** (dos copias, una después de otra).
- Log: $\langle T_i, X, V_1, V_2 \rangle$; commit = registro de commit en almacenamiento estable.
- **Steal** $\Rightarrow$ undo; **no-force** $\Rightarrow$ redo. Los motores: steal + no-force.
- **WAL**: log antes que la página; todo el log antes de responder el commit.
- Libro: redo de todo desde el checkpoint, undo de la undo-list. Idempotente.
- **ARIES**: LSN, pageLSN, prevLSN; TT y DPT; CLR con undoNextLSN; fuzzy checkpoints; analysis, redo (tres excepciones), undo por el mayor LSN.
- PostgreSQL: WAL solo de redo (MVCC + `pg_xact`), LSN en bytes, full-page writes, `pd_lsn`, `synchronous_commit`, `CHECKPOINT_REDO`.
- Disco perdido: backup + log archivado (PITR); one-safe, two-safe, two-very-safe.

Para la próxima clase

- Laboratorio: el WAL de PostgreSQL (`pg_walinspect`, `pageinspect`, una caída con `db-crash` y `synchronous_commit`).
- P5 (`quipu`: transacciones) se entrega y se presenta en la semana 16.
- Leer Silberschatz, secs. 20.1–20.5, 21.2–21.4, 23.1–23.2 y 23.8, y la nota 16 de CS186 (*Distributed Transactions*).
- Correr `make test` en `code/week-15` y leer los apuntes de la semana.

Pregunta

Un envío se registra en Lima con un commit y, un milisegundo después, se cae el servidor de Lima. El servidor de respaldo está en Arequipa. ¿Qué tiene que haber pasado antes de que el commit respondiera para que el envío no se pierda? ¿Qué cuesta eso en cada commit?