

UNIVERSIDAD NACIONAL DE INGENIERÍA

Bases de Datos

Bases de datos distribuidas, NoSQL y cierre del curso

Transacciones y sistemas distribuidos

Gerson Garrido

Semana 16

Hoy

Sesión A: bases de datos distribuidas

1. Arquitecturas: de una máquina a muchas
2. Particionamiento y skew
3. Particionamiento en PostgreSQL y pruning
4. Replicación síncrona y asíncrona
5. Transacciones distribuidas y 2PC
6. Consensus y CAP, en breve

Sesión B: NoSQL, MapReduce y cierre

7. Key-value y document stores (opcional)
8. MapReduce (opcional)
9. El recorrido del curso
10. Repaso para el final

Lectura

Silberschatz, secs. 20.1–20.5 (panorama), 21.2–21.4, 23.1–23.2 y, en breve, 23.8. CS186, *Distributed Transactions*. Además: CS186 *Parallel Query Processing* y *Consensus*; CMU 15-445 (Fall 2025), *Distributed Database Systems I* y *II*; PostgreSQL, *Table Partitioning*, *High Availability…* y *PREPARE TRANSACTION*; Jepsen, *Consistency Models*. Sesión B: secs. 10.2–10.3 (opcionales) y CS186 *NoSQL* y *MapReduce and Spark* (opcionales); para el repaso, los *Summary* 12.7, 13.8, 14.11, 15.9, 16.7, 17.11, 18.11 y 19.11. Además: PostgreSQL *JSON Types*; calendario de CMU 15-445.

SECCIÓN 01

Arquitecturas

01

Chasqui crece

- Agencias en todo el país, millones de eventos y empleados que no pueden quedarse sin sistema si se cae un servidor.

Capacidad

No caben en una máquina: **particionar**.

Disponibilidad

Si una máquina cae, otra sigue: **replicar**.

Latencia

Los datos de Iquitos, cerca de Iquitos.

El precio

Particionar obliga a coordinar transacciones; replicar, a decidir qué pasa cuando las copias no coinciden.

De una máquina a muchas

- **Centralizada**: un DBMS en una computadora (todo el semestre). PostgreSQL: un proceso por conexión, pero un solo buffer pool y un solo WAL.
- **Cliente-servidor**: la aplicación manda consultas; el servidor hace todo.
- **Paralela**: muchas máquinas en un centro de datos, red rápida, rendimiento.
- **Distribuida**: sitios en ciudades distintas, red lenta y menos confiable, disponibilidad, latencia, autonomía.

Speedup

Mismo problema, sistema N veces más grande: T_S/T_L. Lineal si vale N.

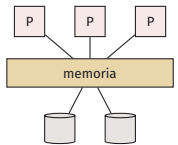
Scaleup

Problema y sistema N veces más grandes: T_S/T_L. Lineal si vale 1.

Casi nunca lineal: costo de arranque, interferencia y skew.

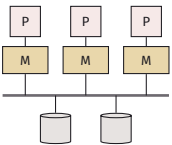
Qué se comparte

memoria compartida



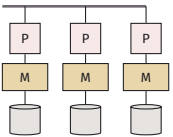
Comunicación barata; no escala más allá de decenas de núcleos.

disco compartido



Si un nodo cae, otro ve los datos; el almacenamiento es el cuello de botella.

nada compartido



Escala a miles de nodos; mover datos por la red es caro.

Idea clave

En shared nothing, casi todo el diseño busca que cada consulta se resuelva con los datos que ya están en un nodo.

Homogéneas y heterogéneas

- Cada sitio atiende **transacciones locales** y participa en **transacciones globales**.

Homogénea

Mismo software en todos los sitios, que se conocen y cooperan: esquema global y un protocolo común de commit.

Heterogénea

Esquemas y software distintos, no diseñados para trabajar juntos (Chasqui compra otra empresa con otro motor).

Autonomía

Un sitio quiere seguir funcionando aunque los demás no respondan. Eso choca con las garantías globales: es la tensión de todo el capítulo.

SECCIÓN 02

Particionamiento

02



Tres maneras de repartir las tuplas

- Fragmentos $r_1, \ldots, r_n$ disjuntos con $r = r_1 \cup \cdots \cup r_n$, uno por nodo.

Round-robin

La i-ésima tupla va a i mód n. Parejo, sin mirar valores.

Hash

La tupla t va a $h(t[A])$, con h en $0..n-1$.

Rango

Vector $[v_0, \ldots, v_{n-2}]$: si $v_i \leq t[A] < v_{i+1}$, va a $i+1$.

- PostgreSQL agrega **lista**: cada partición declara sus valores (las agencias de una zona).

Qué consulta va a qué nodo

Acceso	Round-robin	Hash de `shipment_id`	Rango de `created_at`
Toda la tabla	4 nodos, parejo	4 nodos, parejo	4 nodos, según los rangos
`shipment_id = 501`	4	1	4
`created_at = ...`	4	4	1
Los envíos de marzo	4	4	1

Regla

Una consulta va a un solo nodo si fija el valor (o el rango) del atributo de particionamiento. Las demás van a todos: bien para leer toda la tabla, un desperdicio para una consulta que devuelve una fila.

Particionar para los joins

- `shipment` y `shipment_event` con la misma función hash sobre `shipment_id`: los eventos del 501 viven con el 501.
- El join es la unión de 4 joins locales.
- Si no: **reparticionar** por la clave del join (la fase de partición del hash join, pero por la red).
- O **replicar** la tabla chica (`agency`, 29 filas) en todos los nodos.

Skew

Data skew

Una partición guarda muchas más tuplas: por valor (los envíos de Lima) o por rangos mal elegidos.

Execution skew

Datos parejos, trabajo no: con particiones por mes, las inserciones caen en el mes actual.

Remedios

- Rangos con la distribución real: una muestra ordenada o un histograma equi-depth (semana 12).
- **Nodos virtuales**: 64 particiones para 4 servidores; mover algunas si uno se sobrecarga.
- **Reparticionamiento dinámico**: partir la que creció, como un nodo de B+ tree.
- **Consistent hashing**: un servidor nuevo mueve $\approx 1/n$ de las claves, no casi todas como $h(k) \bmod n$.

SECCIÓN 03

Particionamiento en PostgreSQL

03



PARTITION BY RANGE

partitions.sql

```sql
CREATE TABLE shipment_by_month (
  LIKE shipment INCLUDING DEFAULTS
                INCLUDING CONSTRAINTS,
  PRIMARY KEY (shipment_id, created_at)
) PARTITION BY RANGE (created_at);

CREATE TABLE shipment_2026_01
  PARTITION OF shipment_by_month
  FOR VALUES FROM ('2026-01-01')
              TO ('2026-02-01');
-- ... una por mes, hasta junio ...
CREATE TABLE shipment_month_default
  PARTITION OF shipment_by_month DEFAULT;

INSERT INTO shipment_by_month
SELECT * FROM shipment;
```

- Dentro de un servidor: la tabla particionada no guarda filas; cada partición es una tabla con su archivo e índices.
- `FROM` incluye, `TO` excluye.
- `DEFAULT`: lo que no cae en ningún rango.
- Repartir particiones en servidores es **sharding**: no lo hace PostgreSQL solo.
- Borrar enero: `DROP` o `DETACH` de una partición, sin `DELETE` masivo.

La clave primaria incluye la clave de particionamiento

- No hay índices globales: cada partición tiene el suyo.
- Solo se garantiza unicidad si los repetidos caerían en la misma partición.

```
psql chasqui_w16

ERROR:  unique constraint on partitioned table must include all partitioning columns
DETAIL:  PRIMARY KEY constraint on table "t_bad" lacks column "created_at" which is
part of the partition key.
```

(El `DETAIL` es una sola línea en la salida real.)

- `tableoid::regclass` dice en qué partición está cada fila (consulta `by_month`): 1813, 1938, 2149, 2054, 2157 y 1889 envíos de enero a junio.
- La `DEFAULT` queda vacía: los datos van de enero a junio de 2026.

Partition pruning

pruning.sql: prune_march

```
EXPLAIN (COSTS OFF) SELECT count(*) FROM shipment_by_month
WHERE created_at >= '2026-03-01' AND created_at < '2026-04-01';
 Aggregate
   ->  Seq Scan on shipment_2026_03 shipment_by_month
         Filter: ((created_at >= '2026-03-01 00:00:00-05'::timestamp with time zone) AND ...
```

pruning.sql: no_prune

```
EXPLAIN (COSTS OFF) SELECT count(*) FROM shipment_by_month
WHERE sender_id = 777;
 Aggregate
   ->  Append
         ->  Seq Scan on shipment_2026_01 shipment_by_month_1
               Filter: (sender_id = 777)
         ...
         ->  Seq Scan on shipment_month_default shipment_by_month_7
               Filter: (sender_id = 777)
```

- Marzo: una partición, 2149 filas de 12 000.
- Sin `created_at` en el `WHERE`: las siete.
- Append $= r_1 \cup \cdots \cup r_7$.

Con hash: la igualdad poda, el rango no

pruning.sql: hash_eq

```
EXPLAIN (COSTS OFF) SELECT * FROM shipment_by_hash WHERE shipment_id = 501;
 Index Scan using shipment_h0_pkey on shipment_h0 shipment_by_hash
   Index Cond: (shipment_id = 501)
```

pruning.sql: hash_range

```
EXPLAIN (COSTS OFF) SELECT count(*) FROM shipment_by_hash WHERE shipment_id BETWEEN 500 AND 510;
 Aggregate
   ->  Append
         ->  Index Only Scan using shipment_h0_pkey on shipment_h0 shipment_by_hash_1
               Index Cond: ((shipment_id >= 500) AND (shipment_id <= 510))
         ...
         ->  Index Only Scan using shipment_h3_pkey on shipment_h3 shipment_by_hash_4
               Index Cond: ((shipment_id >= 500) AND (shipment_id <= 510))
```

- Cuatro particiones parejas: 2977, 3022, 3053 y 2948 envíos. El hash desordena los valores a propósito; dentro de cada partición, el índice sí sirve para el rango.

Pruning en la ejecución

pruning.sql: prune_runtime

```
SET plan_cache_mode = force_generic_plan;
PREPARE since_day(timestamptz) AS
  SELECT count(*) FROM shipment_by_month
  WHERE created_at >= $1;
EXPLAIN (ANALYZE, COSTS OFF, TIMING OFF, SUMMARY OFF)
EXECUTE since_day('2026-06-15');
 Aggregate (actual rows=1.00 loops=1)
   Buffers: shared hit=24
   ->  Append (actual rows=854.00 loops=1)
         Buffers: shared hit=24
         Subplans Removed: 5
         ->  Seq Scan on shipment_2026_06 shipment_by_month_1
               (actual rows=854.00 loops=1)
               Filter: (created_at >= $1)
               Rows Removed by Filter: 1035
         ->  Seq Scan on shipment_month_default shipment_by_month_2
               (actual rows=0.00 loops=1)
```

(Las líneas `(actual rows=...)` van en la misma línea del `Seq Scan` en la salida real.)

- Con `$1`, el plan genérico no sabe qué meses hacen falta.
- El ejecutor sí, al empezar: `Subplans Removed: 5`.
- Quedan junio (854 envíos desde el 15) y la `DEFAULT`.
- Sin `force_generic_plan`, las primeras 5 ejecuciones se planifican con el valor real.

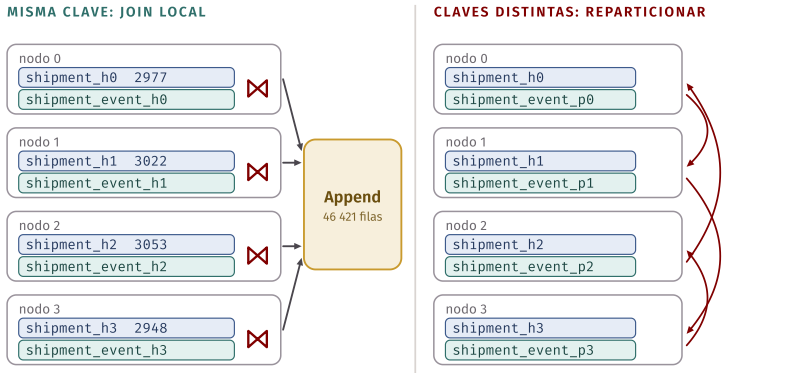
El join partición contra partición

pruning.sql: partitionwise

```
SET enable_partitionwise_join = on;
EXPLAIN (COSTS OFF)
SELECT s.status, count(*)
FROM shipment_by_hash s
  JOIN shipment_event_by_hash e
    ON e.shipment_id = s.shipment_id
GROUP BY s.status;
 HashAggregate
   Group Key: s.status
   ->  Append
         ->  Hash Join
               Hash Cond: (e_1.shipment_id = s_1.shipment_id)
               ->  Seq Scan on shipment_event_h0 e_1
               ->  Hash
                     ->  Seq Scan on shipment_h0 s_1
         ->  Hash Join
               Hash Cond: (e_2.shipment_id = s_2.shipment_id)
               ->  Seq Scan on shipment_event_h1 e_2
               ->  Hash
                     ->  Seq Scan on shipment_h1 s_2
         ...
```

- `shipment_event_by_hash`: mismo hash, mismo módulo 4.
- Cuatro hash joins pequeños y un `Append`.
- Si cada partición estuviera en otro servidor, cada uno haría su join sin mandar nada.
- Viene apagado: planificar join por join cuesta más.
- Resultado igual al del join normal: 46 421 filas.

Misma clave o reparticionar



Skew, medido

pruning.sql: by_zone

```
         partition          |  n   | pct
----------------------------+------+------
 shipment_lima              | 3965 | 33.0
 shipment_sur               | 3147 | 26.2
 shipment_norte             | 2685 | 22.4
 shipment_centro_oriente    | 2203 | 18.4
(4 rows)
```

Lima: 3 de 29 agencias, un tercio de los envíos.

pruning.sql: insert_skew

```
     partition     | n
-------------------+----
 shipment_2026_06 | 50
(1 row)
```

50 envíos nuevos de fines de junio: todos en junio.

Error común

Particionar esperando que todo se acelere. Solo ganan las consultas que filtran por el atributo de particionamiento; las demás leen todas las particiones.

SECCIÓN 04

Replicación

04



Copias de los mismos datos

- n copias: sobrevive a $n - 1$ caídas; las lecturas se reparten.
- El precio está en las escrituras: cada cambio tiene que llegar a todas.
- **Primary-backup**: una copia principal recibe las escrituras; las réplicas aplican los cambios en el mismo orden. Failover: promover una réplica.
- ¿Qué se manda? El log. Una réplica es un servidor en recuperación permanente.

Síncrona (two-very-safe)

El commit espera a la réplica. No se pierde nada al hacer failover; cada commit paga un viaje por la red.

Asíncrona (one-safe)

El commit responde con el WAL local. Rápida; la réplica va atrás y un failover puede perder los últimos commits.

Streaming replication en PostgreSQL

- `make cluster`: `initdb` en el puerto 5416; `pg_basebackup -R` y la copia arranca en el 5417 como **hot standby**.

primario, puerto 5416

```
$ psql -h /tmp -p 5416 -U postgres -c "SELECT application_name, client_addr, state, sync_state, sent_lsn, replay_lsn FROM pg_stat_replication"
 application_name | client_addr |   state   | sync_state | sent_lsn  | replay_lsn
------------------+-------------+-----------+------------+-----------+-----------
 replica1         |             | streaming | async      | 0/4000000 | 0/4000000
```

réplica, puerto 5417

```
$ psql -h /tmp -p 5417 -U postgres -c "SELECT pg_is_in_recovery(), pg_last_wal_replay_lsn()" -c "CREATE TABLE x (a integer)"
 pg_is_in_recovery | pg_last_wal_replay_lsn
-------------------+------------------------
 t                 | 0/4000000

ERROR:  cannot execute CREATE TABLE in a read-only transaction
```

- Asíncrona por defecto. Los LSN de la semana 15 miden cuánto va atrás la réplica.

Síncrona: sin réplica, no hay commit

primario

```
ALTER SYSTEM SET synchronous_standby_names = 'replica1';
SELECT pg_reload_conf();
 application_name |   state   | sync_state
------------------+-----------+------------
 replica1         | streaming | sync
```

Se detiene la réplica y se inserta una fila. El `INSERT` no termina. Desde otra sesión:

otra sesión

```
 pid  | wait_event_type | wait_event | state  |                    query
------+-----------------+------------+--------+-----------------------------------------------
 7840 | IPC             | SyncRep    | active | INSERT INTO replica_demo VALUES (1, 'sin réplica')
```

Y la fila todavía no se ve: `SELECT * FROM replica_demo` da `(0 rows)`.

Se cancela la espera

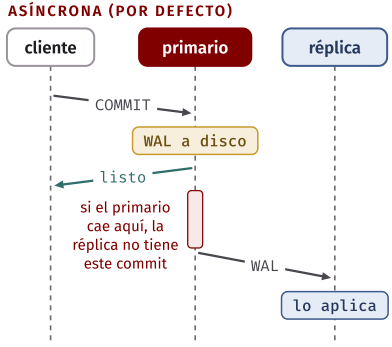
la sesión del INSERT, después de pg_cancel_backend(7840)

```
WARNING:  canceling wait for synchronous replication due to user request
DETAIL:  The transaction has already committed locally, but might not have been
replicated to the standby.
INSERT 0 1
```

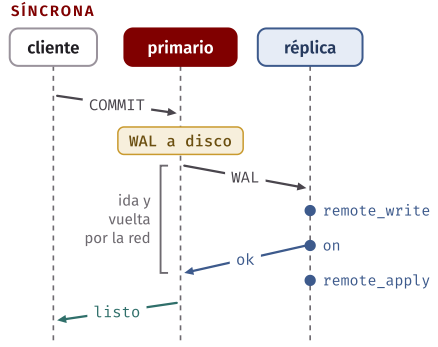
(El `DETAIL` es una sola línea en la salida real.)

- `WARNING`, no `ERROR`: la transacción hizo commit en el primario y la fila aparece.
- Lo que no se cumplió: la promesa de estar en la réplica.
- Sin réplica, el primario deja de confirmar commits. Por eso se configuran varias candidatas: `'ANY 1 (r1, r2)'`.

Commit asíncrono y síncrono



el commit fue rápido, pero la réplica va atrás



nada se pierde en un failover, pero cada commit espera la red; sin réplica, no hay commits

Réplicas y consistencia

- `synchronous_commit`: `remote_write`, `on` (por defecto), `remote_apply` (la réplica ya lo muestra), `local`, `off`.
- Con réplicas asíncronas, garantías intermedias: **read your writes**, **monotonic reads**, **eventual consistency**.
- Detalle real: la réplica no arranca si su `max_prepared_transactions` es menor que el del primario.

Error común

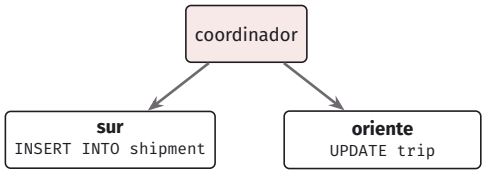
Tomar una réplica como un backup. Un `DELETE FROM shipment` sin `WHERE` llega a la réplica en milisegundos. Para eso están el backup y el point-in-time recovery (semana 15).

SECCIÓN 05

Transacciones distribuidas y 2PC

05

Un envío de Cusco a Iquitos



las dos partes, o ninguna

- Cada sitio: su log, su lock manager, su recuperación.
- Cada uno sabe hacer commit atómico de su parte.
- Falta un protocolo que los ponga de acuerdo.

Por sitio (sec. 23.1)

Transaction manager: log y concurrencia locales. **Coordinator**: inicia, reparte y termina las transacciones que nacen ahí.

Fallos nuevos

Sitio caído, mensaje perdido, enlace caído, **partición de la red**. Desde afuera, igual: alguien no responde.

Two-phase commit

Fase 1: votación

1. El coordinador fuerza `<prepare T>` y manda `prepare T`.
2. Un participante que no puede: `<no T>`, responde `abort T`.
3. Uno que puede: fuerza `<ready T>` con todos los registros de *T* y responde `ready T`.

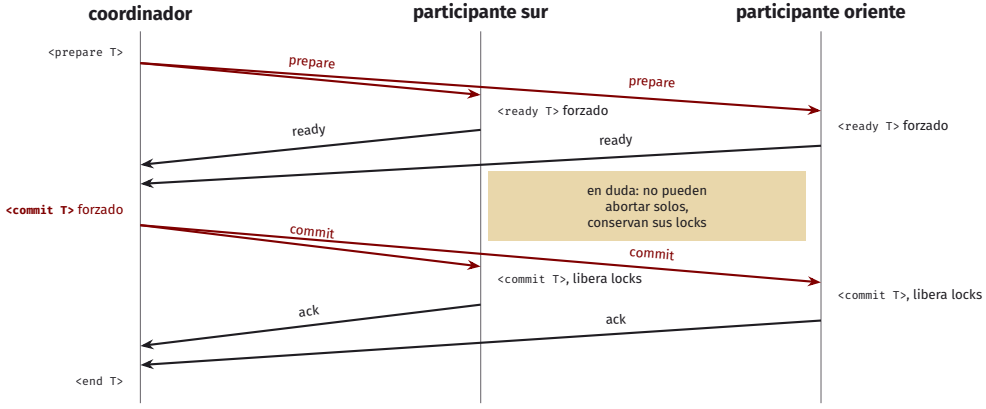
Fase 2: decisión

4. Todos `ready`: fuerza `<commit T>`. Si no (o timeout): `<abort T>`.
5. Manda la decisión; cada participante la anota, termina, libera locks y responde ack.
6. Con todos los acks: `<end T>`.

El instante de la decisión

Cuando `<commit T>` (o `<abort T>`) llega al almacenamiento estable del coordinador. Después de eso, no hay vuelta atrás.

2PC en el tiempo



Con n participantes: $4n$ mensajes y varias escrituras forzadas. Un commit local es un solo `fsync`.

Cuando falla un participante

- Cae antes de votar: el coordinador vence el timeout y aborta.

Al volver, revisa su log para cada T:

En su log	Qué hace
`<commit T>`	redo(T)
`<abort T>`	undo(T)
`<ready T>` sin decisión	en duda: pregunta al coordinador, mantiene los locks
ningún registro de control	no votó $\Rightarrow$ el coordinador no pudo hacer commit: undo(T)

Silberschatz

Escribir `<ready T, L>`, con L la lista de locks de T, para recuperarlos rápido al volver.

Cuando falla el coordinador: el bloqueo

Los participantes activos intentan decidir entre ellos:

- Alguno tiene `<commit T>` o `<abort T>` $\Rightarrow$ esa es la decisión.
- Alguno no tiene `<ready T>` $\Rightarrow$ no votó que sí $\Rightarrow$ se puede abortar.
- Todos tienen `<ready T>` y nadie sabe la decisión $\Rightarrow$ esperar: el coordinador pudo haber escrito `<commit T>` justo antes de caer.

El problema de 2PC

Mientras esperan, mantienen los locks de *T*: un viaje del oriente queda inaccesible porque cayó una máquina en el sur. Es un problema de **disponibilidad**.

Presumed abort

- Observación: si el coordinador no tiene registro de *T*, la respuesta puede ser siempre abort.
- El coordinador no fuerza `<abort T>`; los participantes tampoco, y no mandan ack de un abort.
- El coordinador olvida una transacción abortada enseguida.
- Solo el commit paga todas sus escrituras forzadas.

Por qué importa

Los abortos llegan en ráfagas (un servidor sobrecargado, un deadlock que se repite): justo cuando no conviene agregar trabajo.

2PL no es 2PC

2PL (semana 14)

Control de concurrencia dentro de un servidor: en qué orden se ven las transacciones. Fase de crecimiento y de contracción de locks.

2PC (hoy)

Atomicidad entre servidores: una transacción repartida hace commit en todos o en ninguno. Fase de votación y de decisión.

Juntos

Cada participante aplica strict 2PL a su parte y suelta sus locks recién con la decisión de la fase 2; 2PC hace que la decisión sea la misma en todos.

`PREPARE TRANSACTION` en PostgreSQL

- Fase 1: `PREPARE TRANSACTION 'gid'`. Fase 2: `COMMIT PREPARED` o `ROLLBACK PREPARED`, desde cualquier sesión.
- El coordinador es externo: la aplicación o un gestor de transacciones.

servidor del curso

```
$ psql -d chasqui_w16 -c "BEGIN" -c "PREPARE TRANSACTION 'demo'"
BEGIN
ERROR:  prepared transactions are disabled
HINT:  Set "max_prepared_transactions" to a nonzero value.
```

Por qué viene apagado

Una transacción preparada y olvidada retiene sus locks indefinidamente, frena a `VACUUM` y sobrevive a los reinicios. Cambiar el parámetro exige reiniciar: usamos el cluster desechable (`max_prepared_transactions=10`).

Una transacción preparada, a mano

prepared.sql en la base oriente

```
BEGIN
UPDATE 1
PREPARE TRANSACTION
        gid         | database |  owner   | xid
--------------------+----------+----------+-----
 manual-1@oriente | oriente  | postgres | 762

   locktype    | relation  | transactionid |       mode       | granted | pid
---------------+-----------+---------------+------------------+---------+-----
 relation      | trip      |               | RowExclusiveLock | t       |
 relation      | trip_pkey |               | RowExclusiveLock | t       |
 transactionid |           |           762 | ExclusiveLock    | t       |

 booked_kg
-----------
    940.00

COMMIT PREPARED
 booked_kg
-----------
    945.00
```

- La sesión ya no está en una transacción, pero la 762 existe.
- Sus locks siguen, con `pid` vacío: ningún proceso la tiene.
- Nadie ve los +5 kg hasta `COMMIT PREPARED`.

Sobrevive a una caída

- Se prepara una transacción y se detiene el cluster desechable (`-m immediate`).

/tmp/w16-primary.log

```
LOG:  database system was not properly shut down; automatic recovery in progress
LOG:  redo starts at 0/3000028
LOG:  invalid record length at 0/402B1D8: expected at least 24, got 0
LOG:  redo done at 0/402B0A8 system usage: CPU: user: 0.00 s, system: 0.00 s, elapsed: 0.00 s
LOG:  recovering prepared transaction 797 from shared memory
```

`pg_prepared_xacts` la sigue mostrando: la promesa del `<ready T>` cumplida.

El gid es único en todo el servidor

Con `sur` y `oriente` en un mismo cluster, el segundo `PREPARE` falla (`already in use`). Cada participante usa `chasqui-1@sur`, `chasqui-1@oriente`, como el branch qualifier de XA.

Un coordinador en Node.js

twopc.mjs (lo esencial)

```
await work(clients);  // sur y oriente
// Fase 1: cada uno fuerza su <ready T>
for (const name of names) {
  const c = clients[name];
  try {
    const g = gidLiteral(c, branchGid(gid, name));
    await c.query(`PREPARE TRANSACTION ${g}`);
    prepared.push(name);
  } catch (err) { allReady = false; break; }
}
// si alguno dijo que no: ROLLBACK PREPARED, 'abort'
// La decisión: este INSERT es el <commit T> forzado
await this.log.query(
  'INSERT INTO decision (gid, outcome)
   VALUES ($1, $2)', [gid, 'commit']);
await this.finish(gid, 'commit', names); // fase 2
```

- El log del coordinador: la tabla `decision` de otra base, en autocommit.
- Presumed abort: solo se anotan commits.
- `PREPARE TRANSACTION` no acepta `$1`: el gid se valida con lista blanca y se escapa.
- `oriente` vota con un constraint trigger diferido.

Cuatro escenarios

node twopc-demo.mjs

```
1. Envío de 35 kg
  sur: ready
  oriente: ready
  decisión: commit (en el log del coordinador)
  sur: commit prepared
  oriente: commit prepared
  end
  resultado: commit

2. Envío de 40 kg (ya no cabe)
  sur: ready
  oriente: no (el viaje 7001 no tiene capacidad:
    1015.00 kg de 1000)
  sur: rollback prepared
  decisión: abort (sin log, presumed abort)
```

node twopc-demo.mjs

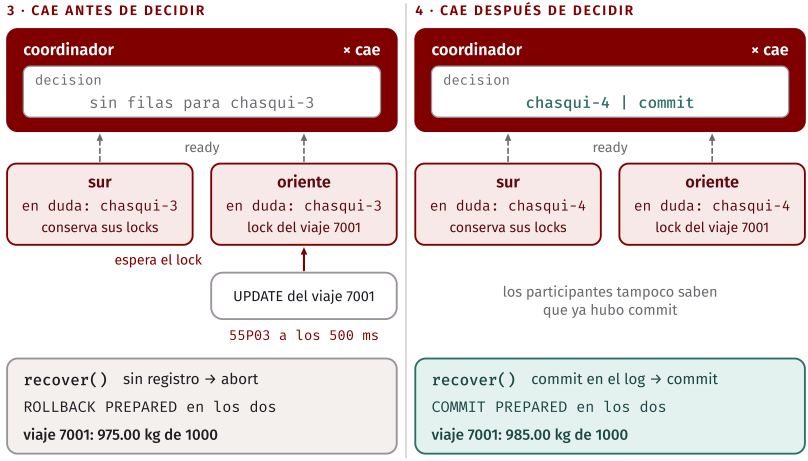
```
3. Envío de 10 kg, el coordinador cae después
   de la fase 1
  sur: ready
  oriente: ready
  el coordinador cayó (prepared)
  otra transacción sobre el viaje 7001: 55P03
    canceling statement due to lock timeout
  recuperación de chasqui-3: sin registro → abort
  sur: rollback prepared
  oriente: rollback prepared

4. ... cae después de decidir
  recuperación de chasqui-4: commit en el log → commit
```

(Las líneas partidas son una sola en la salida real; se omiten algunas líneas.)

Entre 3 y 4, la diferencia es una fila en el log del coordinador.

Escenarios 3 y 4: en duda, con los locks



SECCIÓN 06

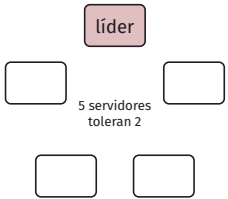
Consensus y CAP

06



Consensus, en breve

- El bloqueo viene de que la decisión vive en una máquina.
- **Consensus** (sec. 23.8): un grupo acuerda un log replicado aunque algunos caigan. Paxos, Raft.



- Decidido = aceptado por una mayoría; dos mayorías siempre se cruzan.
- $2f + 1$ servidores toleran f caídas.
- Raft: un líder agrega al log y replica; si cae, se elige otro con todo lo confirmado.

2PC sobre consensus

Coordinador y participantes como grupos replicados: la caída de una máquina ya no bloquea. Spanner, CockroachDB.

CAP

- Copias en Lima e Iquitos; se corta la fibra. Un empleado en Iquitos registra un envío.

Aceptarlo (A)

Disponible en los dos lados; las copias divergen.

Rechazarlo (C)

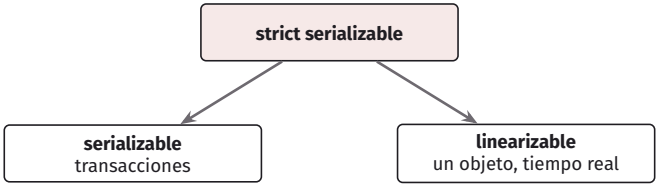
Una sola versión (linearizable); Iquitos no escribe.

- Durante una partición (**P**), no se puede tener **C** y **A**. Sin partición sí, pero la consistencia se paga en latencia.
- PostgreSQL con réplica síncrona eligió C; la réplica asíncrona, A para leer.

Error común

La C de CAP no es la de ACID. ACID: respetar las restricciones. CAP: que las réplicas respondan como si hubiera una sola copia.

Linearizable, serializable, strict serializable



- **Serializable**: igual a alguna ejecución en serie (semanas 13–14). No habla del tiempo real.
- **Linearizable**: cada operación sobre un objeto parece ocurrir en un instante entre su inicio y su fin. Es la C de CAP.
- **Strict serializable**: las dos cosas. Lo que espera un usuario.
- Réplica asíncrona: lee un snapshot consistente, pero puede no ver lo que ya terminó en el primario.

SECCIÓN 07

NoSQL (opcional)

07



Por qué aparecieron

- Mediados de los 2000: millones de usuarios, operaciones simples (leer o escribir por clave), crecer con máquinas baratas.
- Se renunció a joins, a transacciones entre muchas filas y, a veces, a la consistencia fuerte.

Key-value

`get`, `put`, `delete` por clave.

Document

Documentos JSON anidados, consultables.

Wide-column

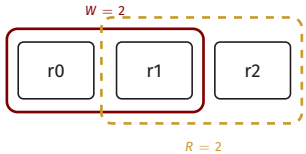
Filas por clave, column families (Bigtable, Cassandra).

Grafos

Nodos y aristas; caminos de largo variable.

Key-value stores y quórums

- El estado de un envío por su código de seguimiento: millones de `get` al día.
- Diseños tipo Dynamo: N réplicas; la escritura espera W, la lectura consulta R.



Regla

Si $R + W > N$, todo conjunto leído se cruza con todo conjunto escrito: la lectura ve la última escritura.

$N = 3$	$R + W$	
$W = 2, R = 1$	3	puede leer justo la réplica vieja
$W = 2, R = 2$	4	siempre ve la última
$W = 3, R = 1$	4	lecturas baratas; una réplica caída frena las escrituras
$W = 1, R = 1$	2	lo más rápido: eventual consistency

Document stores: el envío 501 como documento

documents.sql: document_501

```
{
    "events": [
        {
            "seq": 1,
            "kind": "received",
            "agency": "Huánuco - Centro"
        },
        ...
        {
            "seq": 4,
            "kind": "delivered",
            "agency": "Pucallpa - Centro"
        }
    ],
    "origin": "Huánuco - Centro",
    "status": "delivered",
    "weight_kg": 1.30,
    "destination": "Pucallpa - Centro",
    "shipment_id": 501
}
```

- Se lee entero, de un nodo, sin joins.
- Pero el nombre de la agencia está copiado en cada evento.

documents.sql: docs_to_rewrite

```
 documents | copies
-----------+--------
      3531 |   7002
```

- Si la agencia 1 cambia de nombre: 3531 documentos, 7002 copias. La anomalía que la normalización evitaba (semana 7).
- `jsonb` (semana 8): tablas y documentos en la misma base, con ACID.

¿Sin esquema? ¿Sin transacciones?

Error común

Decir que NoSQL "no tiene esquema" o "no tiene transacciones".

- El esquema existe: el código espera un campo `events` con objetos que tienen `seq`.
- Cambia dónde se verifica: **schema-on-write** (`CREATE TABLE`) frente a **schema-on-read** (en cada aplicación que lee).
- Muchos sistemas garantizan atomicidad por documento o por clave; varios ya ofrecen transacciones entre varios documentos.

SECCIÓN 08

MapReduce (opcional)

08



El modelo

- `map(clave, valor)` emite pares (k, v).
- **Shuffle** (lo hace el sistema): junta los pares por k y los manda al reducer $\mathrm{hash}(k)$ mód R.
- `reduce(k, valores)` produce el resultado de k.

pseudocódigo

```
map(evento):
  si evento.kind = 'delivered':
    emit(evento.city, 1)

reduce(ciudad, unos):
  return suma(unos)
```

el mismo cálculo en SQL

```sql
SELECT a.city, count(*)
FROM shipment_event e
  JOIN agency a
    ON a.agency_id = e.agency_id
WHERE e.kind = 'delivered'
GROUP BY a.city;
```

- `map` = σ y Π; shuffle = agrupar; `reduce` = γ.
- **Combiner**: un reduce local antes del shuffle. Solo si la función es asociativa y conmutativa.

MapReduce sobre los eventos de Chasqui

```
PGDATABASE=chasqui_w16 node mapreduce.mjs

46421 eventos

sin combiner: 11115 pares emitidos por map, 11115 enviados
  en el shuffle (1373 + 7407 + 2335)
  Lima       3632
  Arequipa   806
  Chiclayo   601
  Ica        599
  Cusco      506

con combiner: 11115 pares emitidos por map, 100 enviados
  en el shuffle (24 + 48 + 28)
  ...
GROUP BY en PostgreSQL:
  Lima       3632
  Arequipa   806
  ...
```

(Cada línea "sin combiner" y "con combiner" es una sola línea en la salida real.)

- Igual al `GROUP BY` (la prueba compara todas las ciudades).
- Combiner: de 11 115 a 100 pares (25 ciudades $\times$ 4 splits).
- Sin combiner, un reducer recibe 7407: el de Lima. Un hash no puede partir una sola clave.

Fallos y lo que vino después

- Miles de máquinas: alguna cae en cada trabajo largo.
- Entrada en un sistema de archivos distribuido y replicado; salida de cada `map` en el disco local.
- Si un worker cae, se vuelve a ejecutar su tarea: `map` y `reduce` son deterministas y sin efectos laterales.

Más allá de MapReduce (sec. 10.4)

Spark guarda los intermedios en memoria y ofrece select, join y group by sobre DataFrames, con un optimizador. La historia termina donde empezó el curso: álgebra relacional y un optimizador.

SECCIÓN 09

Cierre del curso

09

El recorrido

- **Parte 1 (semanas 1–5).** Relaciones, claves, álgebra relacional, SQL con multiconjuntos y `NULL`, joins, agregación, SQL desde Node.js, triggers, recursión, window functions.
- **Parte 2 (semanas 6–8).** E-R, dependencias funcionales, BCNF y 3NF, `jsonb`.
- **Parte 3 (semanas 9–12).** Páginas, buffer pool, B+ trees, costo en bloques, joins, optimizador. `quipu`: P3 y P4.
- **Parte 4 (semanas 13–16).** Serializabilidad, 2PL, MVCC, WAL y ARIES, y lo que cambia al distribuir. `quipu`: P5.

Las ideas se repiten

El log de la semana 15 es lo que se manda a las réplicas. El hash join es el reparticionamiento y el shuffle. El histograma elige los rangos. Strict 2PL es lo que aplica cada participante de 2PC.

Lo que no vimos

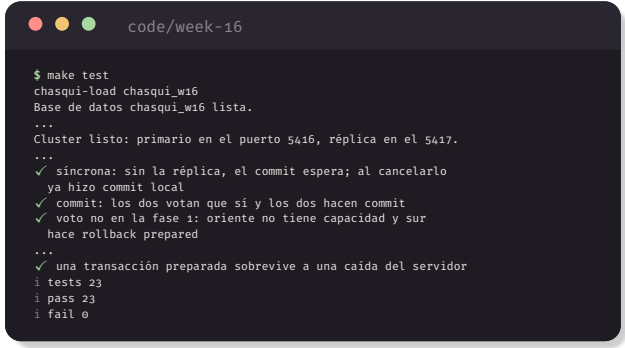
- Column stores y analítica (Silberschatz, cap. 11).
- Procesamiento de streams.
- Bases de datos de grafos en serio.
- Seguridad y control de acceso: `GRANT`, roles, row-level security.
- PostgreSQL en producción: monitoreo, ajuste de `VACUUM`, backups automáticos, actualizaciones de versión, connection poolers.

Repaso para el final

- Todo el curso, con énfasis en las partes 3 y 4.
- Tipos de ejercicio:
  - insertar en un B+ tree: splits y altura;
  - costo de un join o de un ordenamiento externo, en bloques;
  - selectividad con un histograma; leer un plan con `EXPLAIN ANALYZE`;
  - schedule serializable por conflictos (grafo de precedencia);
  - 2PL y deadlocks; anomalías de cada nivel de aislamiento;
  - recuperar un log con ARIES.
- El final de práctica (`exams/final-practica.md`): en papel, con el reloj corriendo.

El código de la semana

- `partitions.sql`, `pruning.sql`, `documents.sql`: sobre `chasqui_w16`.
- `make cluster`: primario (5416) y réplica (5417) desechables.
- `prepared.sql`, `twopc.mjs`, `twopc-demo.mjs`, `mapreduce.mjs`.
- `make test` borra el cluster al final.



(Las líneas partidas son una sola en la salida real.)

Resumen

- Distribuir: **capacidad** (particionar), **disponibilidad** (replicar), **latencia**. Shared nothing escala; lo caro es la red.
- Round-robin, hash, rango (y lista). Una consulta va a un nodo solo si fija el atributo de particionamiento. Co-partitioning hace locales los joins. Skew de datos y de ejecución.
- PostgreSQL: `PARTITION BY`, partition pruning al planificar y al ejecutar, join por particiones. Sharding es otra cosa.
- Replicación síncrona (no pierde, pero sin réplica no hay commit) o asíncrona (rápida, va atrás). Una réplica no es un backup.
- **2PC**: ready forzado y sin vuelta atrás; la decisión es un registro en el log del coordinador; si el coordinador cae, bloqueo. Presumed abort.
- `PREPARE TRANSACTION`, `COMMIT PREPARED`: sobreviven a caídas y conservan locks; gid único por servidor. 2PL $\neq$ 2PC.
- Consensus: $2f + 1$ toleran f. CAP: C o A durante una partición; la C de CAP no es la de ACID.
- Opcional: key-value ($R + W > N$), document stores, MapReduce $=$ `GROUP BY` con shuffle.

Para la próxima clase

- Laboratorio: presentación de **P5** (los cuatro escenarios del lock manager y del log de `quipu`). Las pruebas de P3, P4 y P5 deben pasar.
- Correr `make test` en `code/week-16` y leer los apuntes de la semana.
- Repasar con los *Summary* de los capítulos 12 a 19 de Silberschatz y resolver el final de práctica.

Pregunta

En 2PC, el coordinador recibe `ready` de todos y cae antes de escribir su decisión. ¿En qué estado quedan los participantes? ¿Por qué no pueden abortar por su cuenta? ¿Qué pasa con sus locks? ¿Qué cambia si el coordinador es un grupo de tres servidores con Raft?