

FIXING LOCK CONTENTION IN OLTP

ANTS AASMA

Nordic PGDay 2026



Hello



About me

- Cybertec Lead Database Consultant
- PostgreSQL contributor
- Performance, HA, extensions, etc.



The problem statement



Wait events to the rescue

```
postgres=# SELECT wait_event_type, wait_event, count(*)
postgres-# FROM pg_stat_activity
postgres-# WHERE application_name = 'pgbench'
postgres-# GROUP BY 1, 2 ORDER BY 3 DESC;
 wait_event_type | wait_event      | count
-----+-----+-----
Lock             | transactionid   |      86
Lock             | tuple           |      13
IO               | walSync         |        1
(3 rows)
```



What's happening

Request A:

```
BEGIN;  
UPDATE acct SET b = b + 15 WHERE id = 1;
```

Request B:



What's happening

Request A:

```
BEGIN;  
UPDATE acct SET b = b + 15 WHERE id = 1;  
  
-- waiting for client
```

Request B:

```
BEGIN;  
UPDATE acct SET b = b - 5 WHERE id = 1  
AND b > 5;  
  
-- Blocks . . .
```



What's happening

Request A:

```
BEGIN;  
UPDATE acct SET b = b + 15 WHERE id = 1;  
  
-- waiting for client  
  
COMMIT;
```

Request B:

```
BEGIN;  
UPDATE acct SET b = b - 5 WHERE id = 1  
AND b > 5;  
  
-- Blocks . . .
```



What's happening

Request A:

```
BEGIN;  
UPDATE acct SET b = b + 15 WHERE id = 1;  
  
-- waiting for client  
  
COMMIT;
```

Request B:

```
BEGIN;  
UPDATE acct SET b = b - 5 WHERE id = 1  
                                AND b > 5;  
  
-- Blocks . . .  
  
-- Update completes  
  
COMMIT;
```

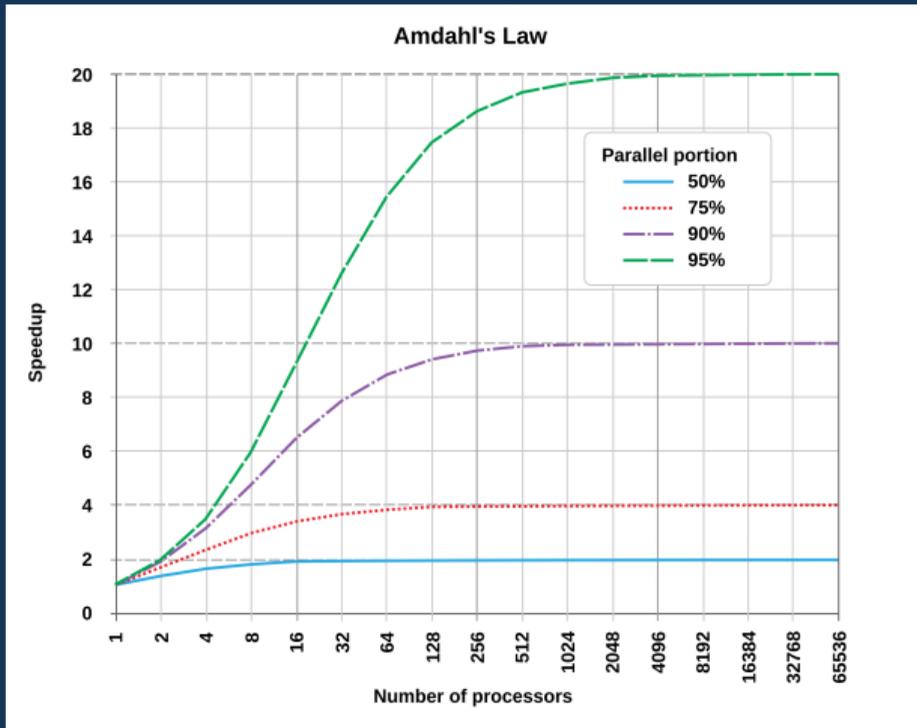


Why this is a big problem

- Double entry accounting requires two account updates for every transaction.
- Some accounts are very hot.
 - “The Bank”
 - “The Payment Processor”
- Throughput is limited by update rate on the hot account.



Amdahl's law strikes again



By Daniels220 at English Wikipedia, CC BY-SA 3.0, <https://commons.wikimedia.org/w/index.php?curid=6678551>



The trouble with SQL



Interactive protocol

```
BEGIN;  
UPDATE accounts .. WHERE act = :a1 ..;  
UPDATE accounts .. WHERE act = :a2 ..;  
INSERT INTO transactions ...;  
SELECT ...;  
COMMIT;
```



Interactive protocol

```
BEGIN;  
UPDATE accounts .. WHERE act = :a1 ..;  
UPDATE accounts .. WHERE act = :a2 ..;  
INSERT INTO transactions ...;  
SELECT ...;  
COMMIT;
```

- Database has no idea what will happen next.



Interactive protocol

```
BEGIN;  
UPDATE accounts .. WHERE act = :a1 ..;  
UPDATE accounts .. WHERE act = :a2 ..;  
INSERT INTO transactions ...;  
SELECT ...;  
COMMIT;
```

- Database has no idea what will happen next.
- Every command is a network roundtrip.



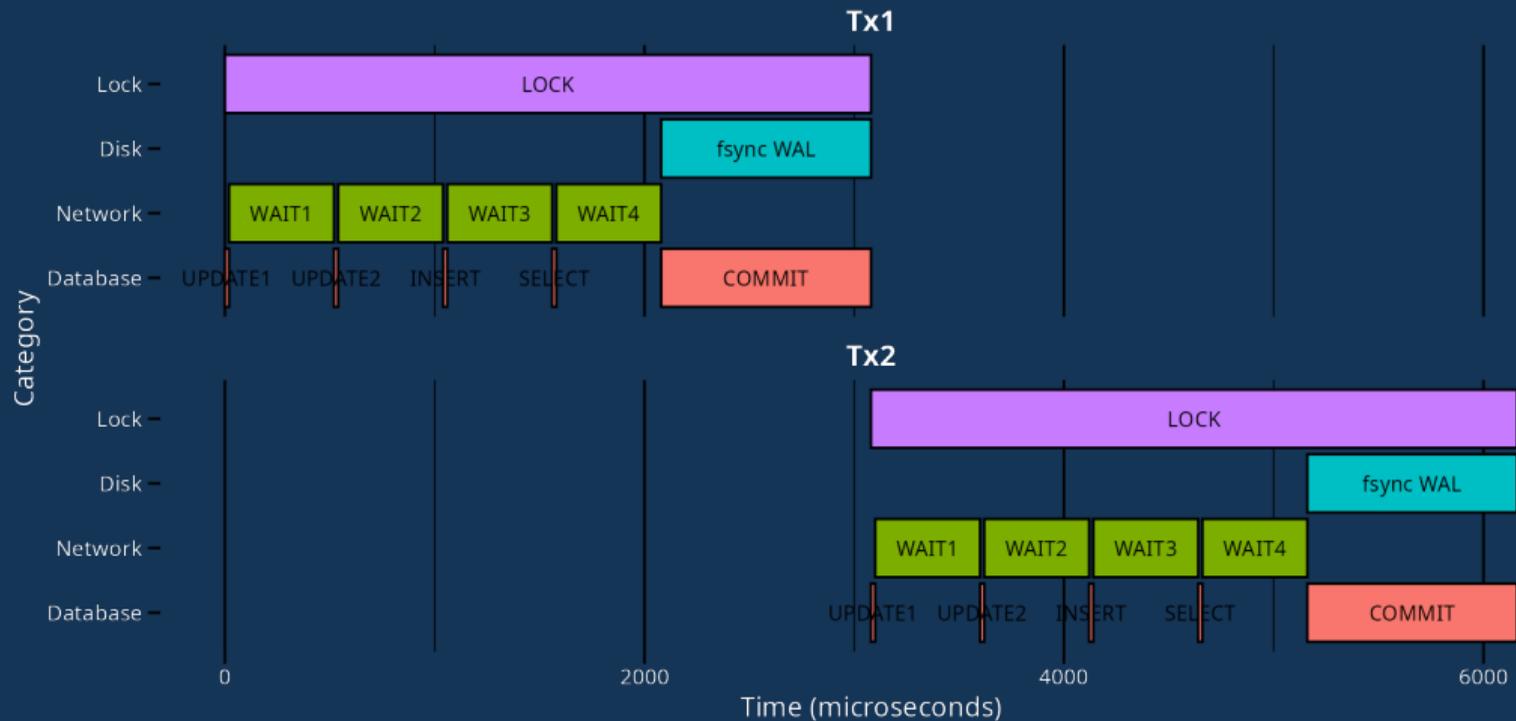
Ballpark numbers

- UPDATE:
 - HOT update using PK: 0.02 ms
- Network roundtrip:
 - Fast network: 0.1 .. 0.2 ms
 - Cloud network: 0.5ms
 - Cloud cross AZ: 0.5 .. 2 ms
 - Cross region: 10 .. 50ms
- Disk roundtrip:
 - Enterprise NVMe: 0.02 ms
 - Good network disk: 0.5ms
 - Samsung SSD: 5ms



What it looks like

Transaction timeline conversational execution



Activity Database Network Disk Lock



Simple optimizations

- Minimize network roundtrip time.
- Do locking actions late to minimize lock hold time.



Taking the logic to the data



Option 1: CTEs

```
WITH update1 AS (UPDATE ... RETURNING ...),
     update2 AS (UPDATE ... RETURNING ...),
     insert1 AS (INSERT ...
                 SELECT .. FROM update1, update2
                 RETURNING ...),
SELECT ... FROM insert1;
```

Will get quite hairy if the logic is more complicated.

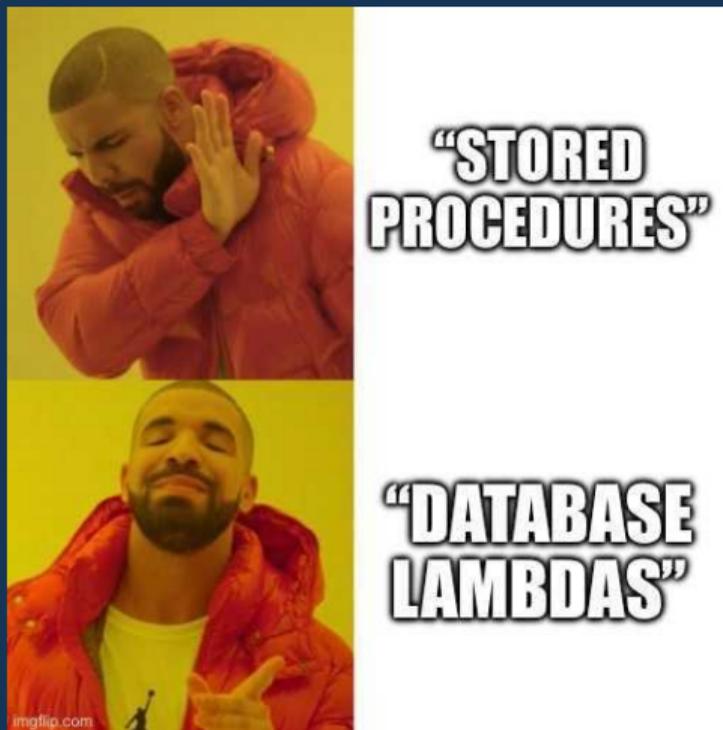


Option 2: Pipeline mode

- Send multiple commands to PostgreSQL in a single roundtrip.
- If any command errors the whole pipeline is cancelled.
- Command must raise errors on failure.
- Very few client drivers support it. (libpq, psycopg)



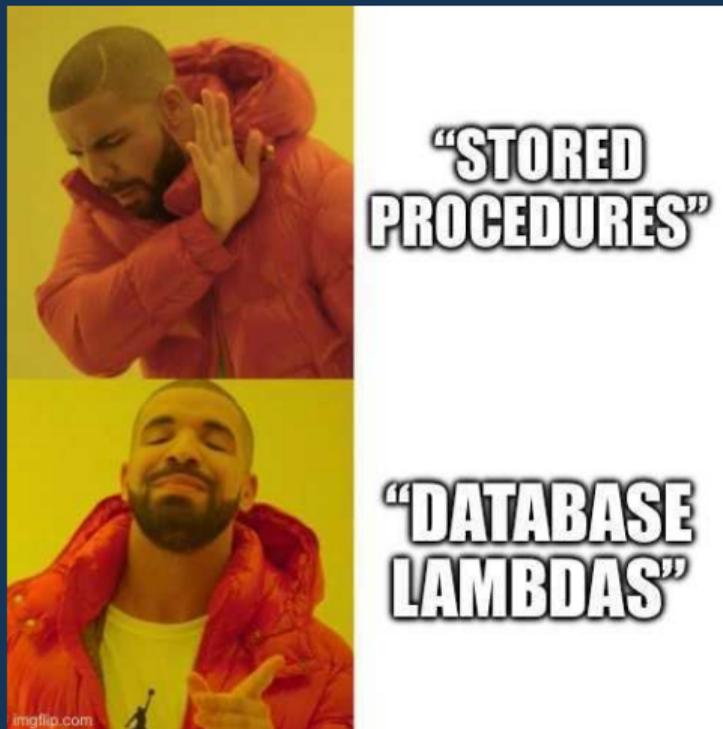
Option 3: Stored procedures



Write the data logic as a single stored function.



Option 3: Stored procedures



Write the data logic as a single stored function.

If you really really hate stored procedures, you can use DO block to achieve the same result.



Stored procedure implementation

```
CREATE FUNCTION transfer(p_src int, p_dest int, p_amount numeric) RETURNS int8 AS $$
DECLARE new_balance numeric; tx_id int8;
BEGIN
    UPDATE accounts SET balance = balance - p_amount
        WHERE id = p_src RETURNING balance INTO new_balance;
    IF new_balance < 0 THEN RAISE EXCEPTION 'Not enough funds in %', p_src; END IF;

    UPDATE accounts SET balance = balance + p_amount
        WHERE id = p_dest RETURNING balance INTO new_balance;
    IF new_balance < 0 THEN RAISE EXCEPTION 'Not enough funds in %', p_dest; END IF;

    INSERT INTO transactions (src, dest, amount)
        VALUES (p_src, p_dest, p_amount)
        RETURNING id INTO tx_id;

    RETURN tx_id;
END;
$$ LANGUAGE plpgsql;
```



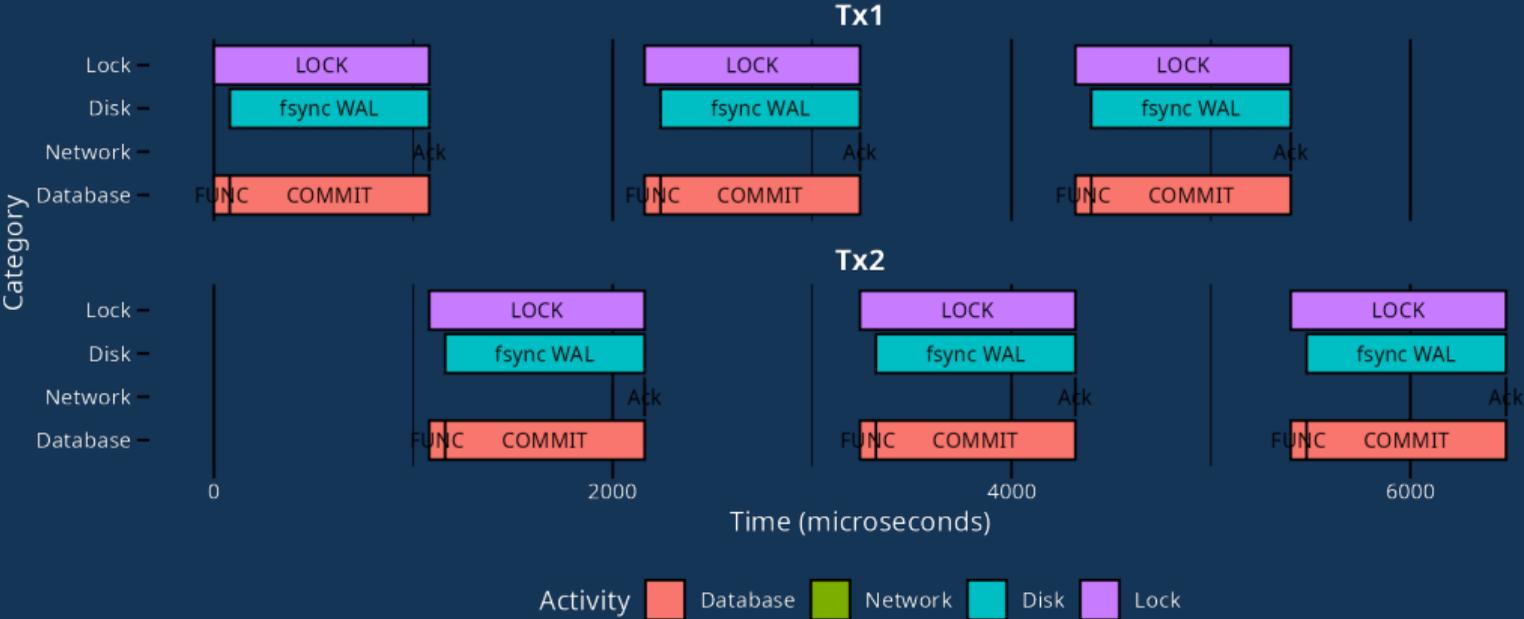
Avoiding all roundtrips

- Send the function call outside explicit transaction.
- Starts implicit transaction in database.
 - If command fails gets rolled back.
 - If command succeeds gets committed automatically.
- Some drivers need special handling for this.
 - psycopg - `autocommit=True`
 - JDBC - `conn.setAutoCommit(true);` (default)



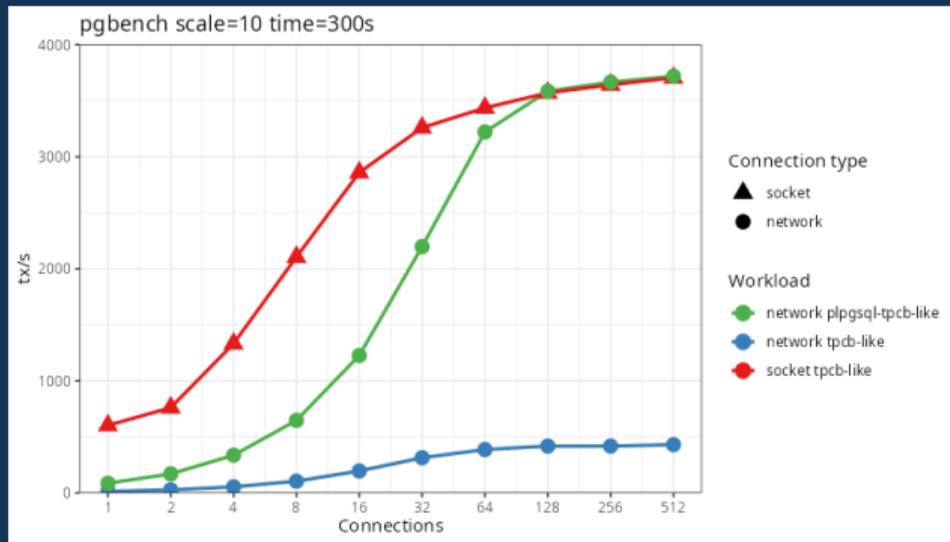
What server side execution looks like

Transaction timeline with server side execution



The benefit

Network has 10ms roundtrip latency.



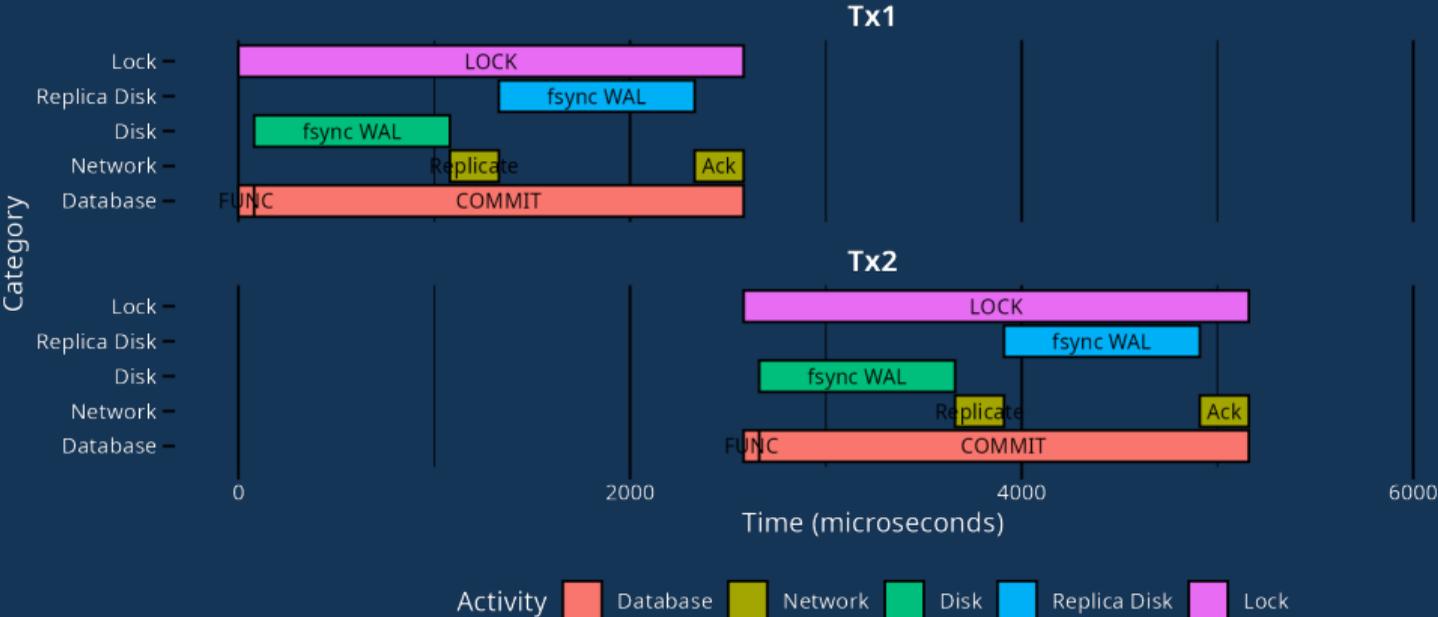
plpgsql-tpcb-like implementation from Hannu Krosing's patch (pgbench - adding pl/pgsql versions of tests)

Tested on Ryzen 9 9900X, Samsung SSD 990 Pro 2TB



Synchronous replication

Transaction timeline for synchronous replication



Avoiding the commit latency



The naughty option

- `synchronous_commit=off` disables WAL flush.
- From 800 tps to 80'000 tps.
- On any crash/failover you will lose transactions.
- Typically less than `wal_writer_delay = 200ms`
- May be ok if you can detect crashes and replay transactions.



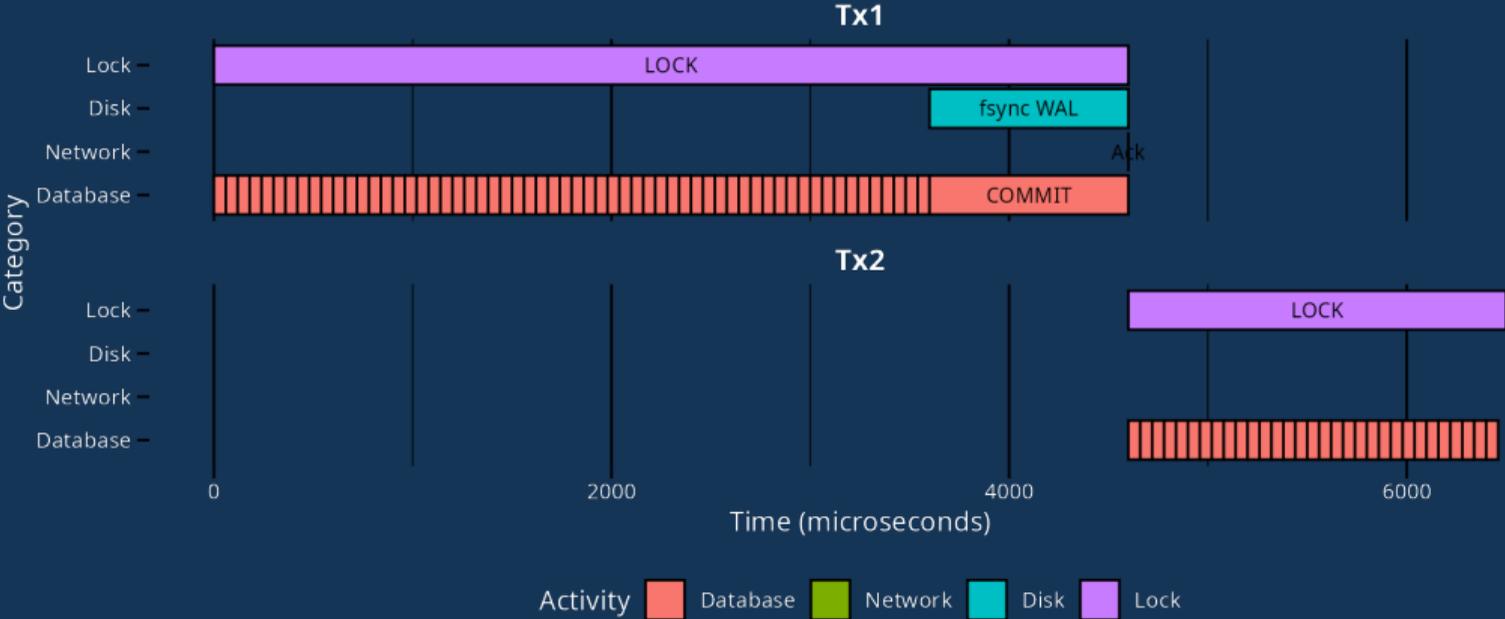
The clever option

- Amortize the durability cost across multiple transactions.
- Collect batch of transactions in application.
- Send to server function for execution.
- Server returns list of result statuses.

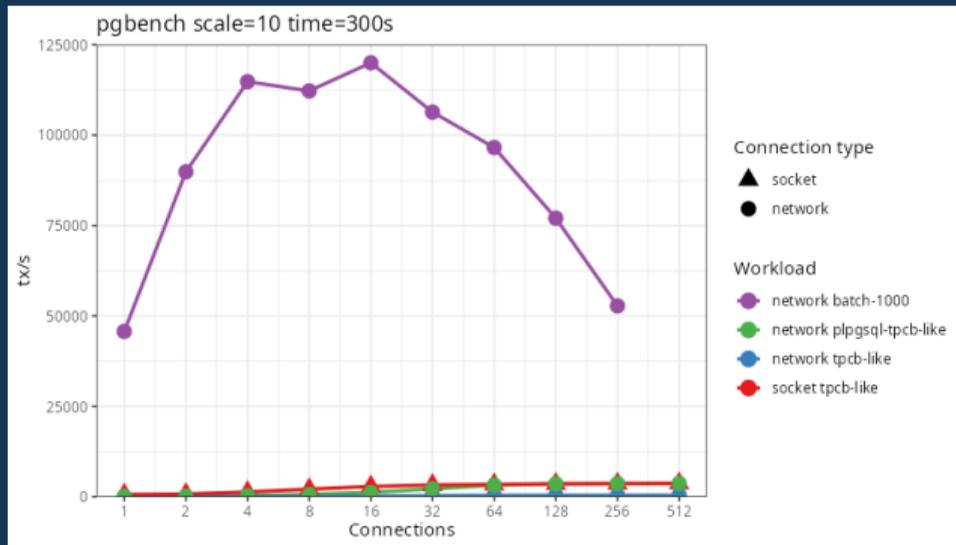


What batching looks like

Transaction Timeline (Faceted by Transaction)



Effect on performance



Downsides

- Complicated work needed to manage batches.
- Need to be careful to limit latency hit.



Transaction isolation



Isolation levels

When two transactions update same row:

READ COMMITTED

1. Block until locker finishes.
2. Look at the new version.
3. Recheck if WHERE condition matches.
4. Update new version.

REPEATABLE READ / SERIALIZABLE

1. Block until locker finishes.
2. If locker committed then raise serialization error.



Contention hell

REPEATABLE READ is almost unusable for contended workloads.

- Any contention will cause concurrent transaction to fail/retry.
- Extremely inefficient use of database resources.
- Optimistic concurrency control and contention do not mix.
- Even table level locks can't save us - requesting a lock acquires a snapshot.



Be careful with correctness

- Read committed allows a single query to see different database states.
- Quite complicated to prove if this affects application correctness.



Possible workaround

Session level advisory locks allow snapshot to be released after locking.

```
-- Block until all accounts available
SELECT pg_advisory_lock(x) FROM unnest(:batch) x ORDER BY x;

BEGIN ISOLATION LEVEL REPEATABLE READ;
    -- Do work ...
COMMIT;

-- Unlock
SELECT pg_advisory_unlock_all();
```



Getting rid of contention



Delta tables

- Calculate account balance based on transaction history.
- Periodically update baseline to limit work needed.
- Only usable when transactions do not depend on each other.



Write skew anomaly

Two tables.

- `flight.seats` is total available seats.
- `booking.booked` is number of seats booked.

Starting condition: only 2 free seats on a flight.



Write skew anomaly

Two tables.

- `flight.seats` is total available seats.
- `booking.booked` is number of seats booked.

Starting condition: only 2 free seats on a flight.

Try to book 2 seats for A:

```
BEGIN;  
SELECT f.seats - SUM(b.booked) available  
FROM flight f JOIN booking b USING (flight_id)  
WHERE flight_id = 123 GROUP BY flight_id;  
  
IF available >= 2 THEN  
  INSERT INTO booking VALUES (123, 'tx A', 2);  
END IF;  
COMMIT;
```



Write skew anomaly

Two tables.

- `flight.seats` is total available seats.
- `booking.booked` is number of seats booked.

Starting condition: only 2 free seats on a flight.

Try to book 2 seats for A:

```
BEGIN;  
SELECT f.seats - SUM(b.booked) available  
FROM flight f JOIN booking b USING (flight_id)  
WHERE flight_id = 123 GROUP BY flight_id;  
  
IF available >= 2 THEN  
  INSERT INTO booking VALUES (123, 'tx A', 2);  
END IF;  
COMMIT;
```

Try to book 2 seats for B:

```
BEGIN;  
SELECT f.seats - SUM(b.booked) available  
FROM flight f JOIN booking b USING (flight_id)  
WHERE flight_id = 123 GROUP BY flight_id;  
  
IF available >= 2 THEN  
  INSERT INTO booking VALUES (123, 'tx B', 2);  
END IF;  
COMMIT;
```



Write skew anomaly

REPEATABLE READ allows for double booking.

Incorrect results.

SERIALIZABLE will fail with serialization error.

Terrible performance with contention.



Sharded accounts

- Usually hot accounts are not close to empty.



Sharded accounts

- Usually hot accounts are not close to empty.
- Split hot accounts into multiple pieces.



Sharded accounts

- Usually hot accounts are not close to empty.
- Split hot accounts into multiple pieces.
- Balance updates between sub-accounts.



Sharded accounts

- Usually hot accounts are not close to empty.
- Split hot accounts into multiple pieces.
- Balance updates between sub-accounts.
- Handle case when one sub-account is empty.



Sharded accounts

- Usually hot accounts are not close to empty.
- Split hot accounts into multiple pieces.
- Balance updates between sub-accounts.
- Handle case when one sub-account is empty.
- Total balance can be calculated on demand.



Far future features for fun



Insights

- We need locking to ensure ordered updates.
- We know the order once we have reserved WAL position.
- Waiting for flush/replication is needed for durability.
- We can speculatively start the next transaction.



Insights

- We need locking to ensure ordered updates.
- We know the order once we have reserved WAL position.
- Waiting for flush/replication is needed for durability.
- We can speculatively start the next transaction.

Detach visibility from durability.



Eventual durability

- Write tx commit releases locks immediately.
 - Write becomes visible to new transactions.
- Write tx commit response waits for durability.
- Read tx keep track of newest tx seen.
- Read tx wait for durability of all seen tx on commit.

Based on Eventual Durability paper from University of Waterloo



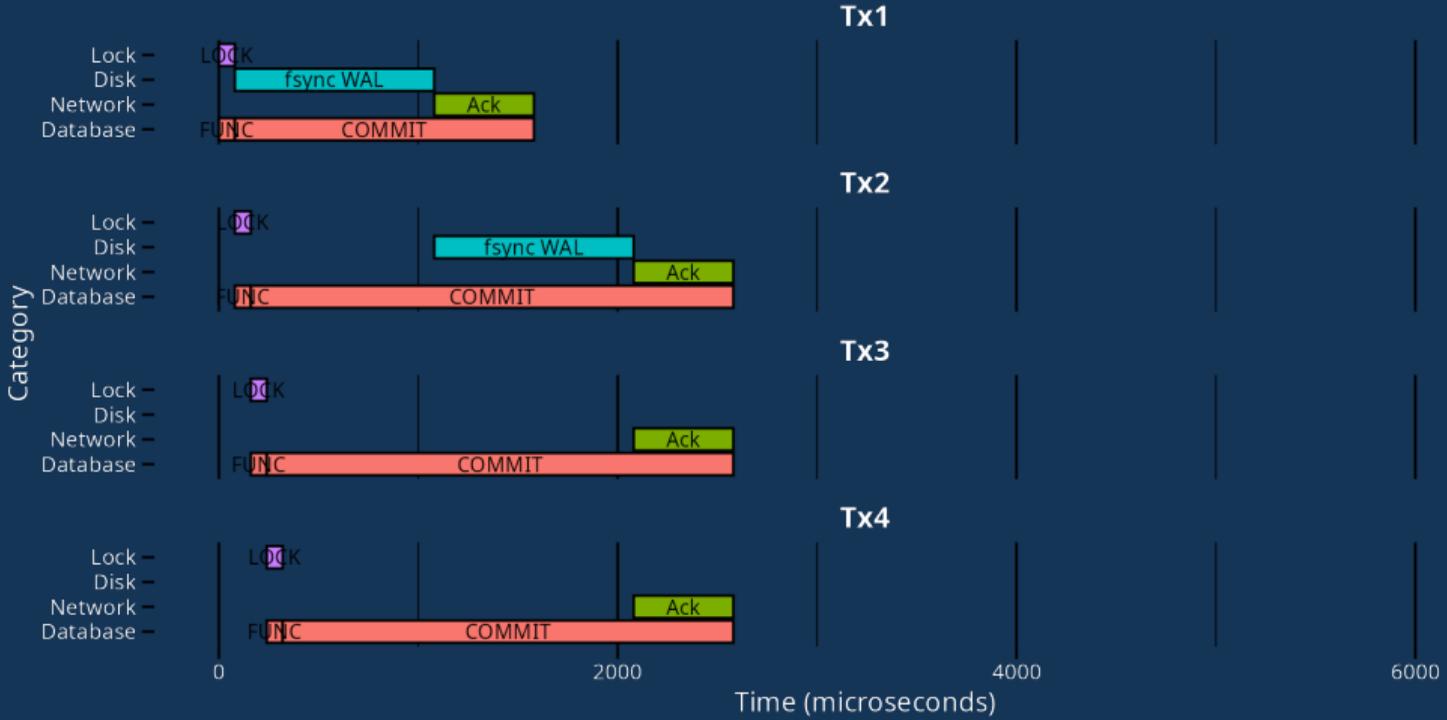
How to implement eventual durability

- Postpone wait for durability after releasing locks.
- When reading, keep track of latest transaction seen.
 - Need mapping xid -> commit LSN.
- On read-only transaction commit, wait for durability of seen LSN.



Eventual durability timeline

Transaction timeline with eventual durability



Activity Database Network Disk Lock



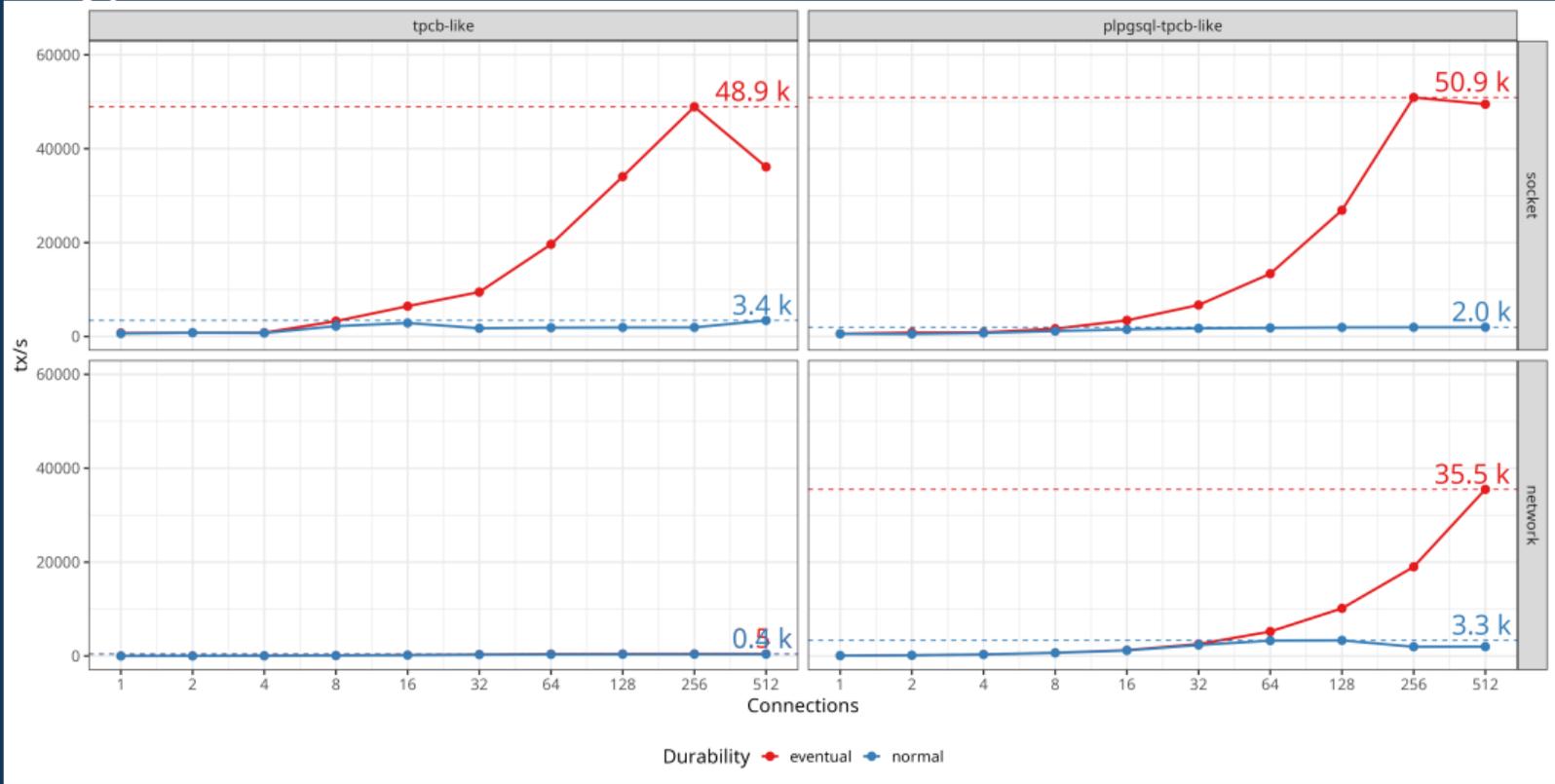
Eventual durability benefit

Almost `synchronous_commit=off` throughput on contended workloads with transactional guarantees.

No client side batching needed.



Prototype!



Recap



What we learned today

- Avoid contended updates if you can.



What we learned today

- Avoid contended updates if you can.
- Figure out if READ COMMITTED is good enough.



What we learned today

- Avoid contended updates if you can.
- Figure out if READ COMMITTED is good enough.
- Minimize transaction length.



What we learned today

- Avoid contended updates if you can.
- Figure out if READ COMMITTED is good enough.
- Minimize transaction length.
- Get rid of network roundtrips using server side functions.



What we learned today

- Avoid contended updates if you can.
- Figure out if READ COMMITTED is good enough.
- Minimize transaction length.
- Get rid of network roundtrips using server side functions.
- Amortize durability delay by batching.



Thank you



Questions?

