PostgreSQL Locking & Concurrency Control

Pavan Deolasee,
Senior Principal Engineer,
13th March 2026

About Me

- Senior Principal Engineer at EDB
- Postgres Distributed PGD (BDR
- Publicity co-chair for SIGMOD 2026
  https://2026.sigmod.org/

- Your co-organizer for this event

What We'll Cover

- Why locking is essential & what problems it solves
- PostgreSQL's locking subsystems (Spinlocks, LWLocks, Heavyweight)
- Deep dive into heavyweight locks
- Lock lifecycle & prepared transactions
- Deadlocks: causes, detection, resolution
- Diagnosing locks with pg_locks & pg_stat_activity
- Avoiding ACCESS EXCLUSIVE locks
- Recent improvements (PostgreSQL 14-17)

The Fundamental Problem

- Shared memory (buffer pool, caches)
- Data files and tables
- Internal data structures
- System catalogs
    - Multiple processes accessing shared resources simultaneously:
    - Without coordination = Chaos

What Locking Provides

- Mutual Exclusion — Only one process modifies critical data at a time
- Data Integrity — Prevents partial updates and corrupted states
- Consistency — Transactions see a coherent view of data
- Isolation — Concurrent operations don't interfere with each other

Problems Locking Solves

Lost Updates

Session A	Session B
SELECT balance FROM accounts WHERE id=1; → 1000	
	SELECT balance FROM accounts WHERE id=1; → 1000
Calculate: 1000 - 200 = 800	Calculate: 1000 - 300 = 700
UPDATE accounts SET balance=800 WHERE id=1;	
	UPDATE accounts SET balance=700 WHERE id=1;

Expected: 500 | Actual: 700 — One update lost!

Dirty Reads

Session A	Session B
BEGIN;	
UPDATE accounts SET balance=0 WHERE id=1;	
	SELECT balance FROM accounts WHERE id=1; → 0
ROLLBACK;	Makes decisions based on invalid data

Session B read data that never actually committed

Inconsistent Reads

Session A	Session B
BEGIN;	
UPDATE accounts SET balance = balance - 500 WHERE id = 'A';	
	SELECT SUM(balance) FROM accounts;
UPDATE accounts SET balance = balance + 500 WHERE id = 'B';	

Session 2 sees $500 missing from the system!

PostgreSQL Locking Subsystems

Three Layers of Locks

Property	Spinlock	LWLock	Heavyweight
Duration	Nanoseconds	Microseconds	Ms to seconds
Wait method	Busy-wait (spin)	Sleep	Sleep + queue
Deadlock detection	No	No	Yes
Modes	Exclusive only	Shared/Exclusive	8 modes
Visible in pg_locks	No	No	Yes
Held across	Instructions	Operations	Transactions

Spinlocks

The fastest, most primitive locks

What are Spinlocks?

- Lowest-level locking primitive
- Implemented using CPU atomic instructions
- Process "spins" (busy-waits) while waiting
- Held for extremely short durations
- No fairness guarantees

Where PostgreSQL Uses Spinlocks

- Protecting LWLock state changes
- Atomic updates to shared counters
- Short critical sections in shared memory
- Buffer header manipulation

LWLocks (Lightweight Locks)

Protecting shared memory structures

What are LWLocks?

- Protect shared memory data structures
- Support shared and exclusive modes
- Waiters sleep (don't spin for long)
- No deadlock detection (programmer must avoid)
- Automatically released at transaction end

Important LWLocks

LWLock	Protects
BufferContent	Individual buffer page contents
WALInsert	WAL buffer insertion
WALWrite	WAL file writes
ProcArray	Process/transaction info array
CLogControl	Commit log (CLOG) buffers

Monitoring LWLock Waits

- WALInsert → increase wal_buffers
- ProcArray → use connection pooling
- BufferMapping → increase shared_buffers
- Common contention issues:

```
SELECT pid, wait_event_type, wait_event, state, query
FROM pg_stat_activity
WHERE wait_event_type = 'LWLock';
```

https://www.postgresql.org/docs/current/monitoring-stats.html#WAIT-EVENT-TABLE

Heavyweight Locks

Locks users interact with regularly

What are Heavyweight Locks?

- Also called "regular locks" or "lock manager locks"
- Protect database objects (tables, rows, etc.)
- Visible in pg_locks
- Support deadlock detection
- Released at transaction end
- Have a lock queue for fairness

Lockable Object Types

locktype	What it locks
relation	Tables, indexes, sequences
tuple	Individual rows
transactionid	Transaction IDs
virtualxid	Virtual transaction IDs
extend	Right to extend a relation
object	Functions, schemas, etc.
advisory	Application-defined locks

Table-Level Lock Modes

Lock Mode	Acquired By
ACCESS SHARE	SELECT
ROW SHARE	SELECT FOR UPDATE/SHARE
ROW EXCLUSIVE	INSERT, UPDATE, DELETE, MERGE
SHARE UPDATE EXCLUSIVE	VACUUM, ANALYZE, CREATE INDEX CONCURRENTLY
SHARE	CREATE INDEX
SHARE ROW EXCLUSIVE	CREATE TRIGGER, some ALTER TABLE
EXCLUSIVE	REFRESH MAT VIEW CONCURRENTLY
ACCESS EXCLUSIVE	DROP, TRUNCATE, VACUUM FULL, most ALTER TABLE

Lock Compatibility Matrix

	AS	RS	RE	SUE	S	SRE	E	AE
ACCESS SHARE								✗
ROW SHARE							✗	✗
ROW EXCLUSIVE					✗	✗	✗	✗
SHARE UPDATE EXCL				✗	✗	✗	✗	✗
SHARE			✗	✗	✓	✗	✗	✗
SHARE ROW EXCL			✗	✗	✗	✗	✗	✗
EXCLUSIVE		✗	✗	✗	✗	✗	✗	✗
ACCESS EXCLUSIVE	✗	✗	✗	✗	✗	✗	✗	✗

Row-Level Locks

- Row locks stored in tuple header (xmax), not in lock table!

Lock Mode	Acquired By	Use Case
FOR KEY SHARE	FK checks	Weakest, allows most operations
FOR SHARE	SELECT FOR SHARE	Blocks updates/deletes
FOR NO KEY UPDATE	UPDATE (non-key cols)	Allows FOR KEY SHARE
FOR UPDATE	DELETE, UPDATE (key cols)	Strongest, blocks all

Explicit Row Locking

```
-- Lock for update (strongest)
SELECT * FROM orders WHERE id = 1 FOR UPDATE;

-- Non-blocking check
SELECT * FROM orders WHERE id = 1 FOR UPDATE NOWAIT;

-- Skip locked rows (great for queues!)
SELECT * FROM orders WHERE status = 'pending'
ORDER BY created_at LIMIT 10
FOR UPDATE SKIP LOCKED;
```

Lock Lifecycle

When locks are acquired, held, and released

Lock Acquisition Timeline

```
BEGIN;
    ↓
SELECT * FROM users WHERE id = 1;
    → ACCESS SHARE on users (acquired)
    ↓
UPDATE users SET name = 'Alice' WHERE id = 1;
    → ROW EXCLUSIVE on users (upgraded)
    → Row lock on tuple (id=1)
    ↓
COMMIT;
    → ALL locks released automatically
```

When Are Locks Released?

No way to release table/row locks early!

Lock Type	Released When
Table locks	Transaction COMMIT or ROLLBACK
Row locks	Transaction COMMIT or ROLLBACK
Advisory locks (xact)	Transaction COMMIT or ROLLBACK
Advisory locks (session)	Explicit unlock or session end

Subtransactions and Locks

Savepoint rollback does NOT release locks!

```
BEGIN;
UPDATE users SET status = 'pending' WHERE id = 1;  -- Lock acquired

SAVEPOINT sp1;
UPDATE users SET status = 'failed' WHERE id = 1;

ROLLBACK TO sp1;  -- Lock STILL HELD!

UPDATE users SET status = 'success' WHERE id = 1;
COMMIT;  -- Now lock released
```

Prepared Transactions

Locks that survive database restart

What Are Prepared Transactions?

- Part of two-phase commit (2PC) protocol
- Transaction is "prepared" but not yet committed
- Survives crashes and restarts
- Waiting for coordinator to say COMMIT or ROLLBACK

```
BEGIN;
UPDATE accounts SET balance = 0 WHERE id = 1;
PREPARE TRANSACTION 'tx_12345';
-- Transaction is now persistent!
```

Prepared Transactions Hold Locks!

```
-- Session 1
BEGIN;
UPDATE accounts SET balance = 0 WHERE id = 1;
PREPARE TRANSACTION 'my_tx';
-- Session can disconnect

-- Session 2 (or after restart!)
UPDATE accounts SET balance = 100 WHERE id = 1;
-- BLOCKS! Waiting for prepared transaction

-- Locks persist even across database restarts!
```

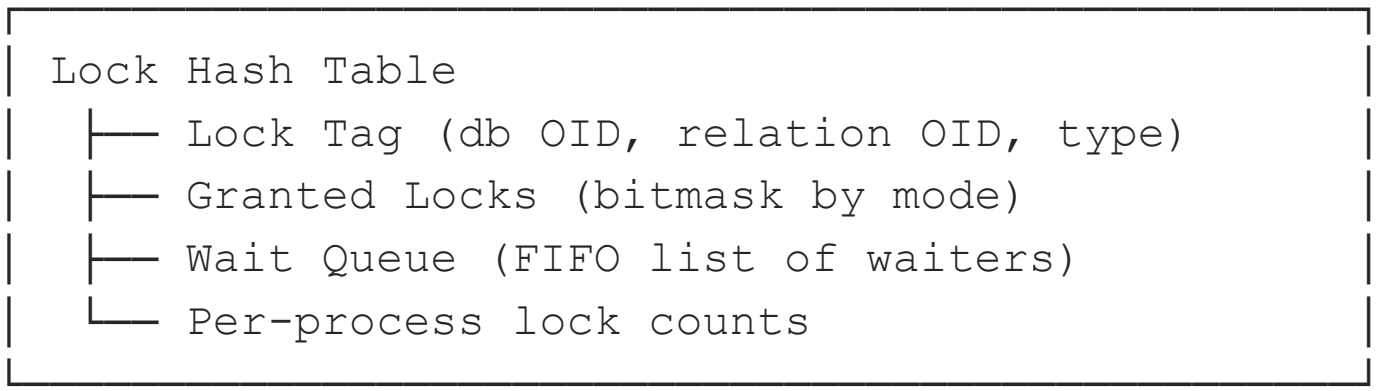
Finding & Resolving Prepared Transactions

```
-- Find prepared transactions
SELECT gid, owner, database, prepared
FROM pg_prepared_xacts;

-- Find old ones (danger!)
SELECT gid, prepared, now() - prepared AS age
FROM pg_prepared_xacts
WHERE prepared < now() - interval '1 hour';

-- Resolve them
COMMIT PREPARED 'my_tx';
-- or
ROLLBACK PREPARED 'my_tx';
```

Lock Manager Internals

Lock Manager Data Structures

Shared Memory Lock Table:



The Wait Queue

Table T with ACCESS SHARE held by Tx1

Queue: [Tx1:AS granted] [Tx2:AE waiting] [Tx3:AS waiting]

- Tx2 must wait for Tx1
- Tx3 must wait for Tx2 (even though compatible with Tx1!)

Why? Prevents starvation of exclusive lockers.

Deadlocks

Deadlock Example

Both waiting for each other → DEADLOCK

Transaction A	Transaction B
UPDATE t SET x=1 WHERE id=1; ✓	
	UPDATE t SET x=2 WHERE id=2; ✓
UPDATE t SET x=1 WHERE id=2;	
	UPDATE t SET x=2 WHERE id=1;

Deadlock Detection

PostgreSQL detects deadlocks automatically:

```
ERROR: deadlock detected
DETAIL: Process 1234 waits for ShareLock on transaction 5678;
        blocked by process 5678.
```

Avoiding Deadlocks

```
-- Strategy 1: Lock in consistent order
SELECT * FROM accounts
WHERE id IN (1, 2)
ORDER BY id
FOR UPDATE;

-- Strategy 2: Lock upfront
BEGIN;
SELECT * FROM accounts WHERE id = 1 FOR UPDATE;
SELECT * FROM accounts WHERE id = 2 FOR UPDATE;
-- Now safe to update in any order
COMMIT;
```

Diagnosing Lock Issues

The pg_locks View

- granted = true : Lock is held
- granted = false : Waiting for lock
- waitstart : When waiting started (PG 14+)
- fastpath : Lock taken via fast path optimization
- Key columns:

```
SELECT locktype, relation::regclass, mode, granted, pid
FROM pg_locks
WHERE relation IS NOT NULL;
```

Finding Blocking Sessions

```
-- Simple: Who is blocked?
SELECT
    pid,
    pg_blocking_pids(pid) AS blocked_by,
    query AS blocked_query
FROM pg_stat_activity
WHERE cardinality(pg_blocking_pids(pid)) > 0;
```

Detailed Blocking Analysis

```
SELECT
    blocked.pid AS blocked_pid,
    blocked.query AS blocked_query,
    blocking.pid AS blocking_pid,
    blocking.query AS blocking_query,
    now() - blocking.xact_start AS blocking_duration
FROM pg_stat_activity AS blocked
JOIN pg_stat_activity AS blocking
  ON blocking.pid = ANY(pg_blocking_pids(blocked.pid));
```

Lock Wait Events

- Lock:relation — Waiting for table lock
- Lock:tuple — Waiting for row lock
- Lock:transactionid — Waiting for transaction
- Common events:

```
SELECT pid, wait_event_type, wait_event, state, query
FROM pg_stat_activity
WHERE wait_event_type IN ('Lock', 'LWLock');
```

Terminating Problem Sessions

```
-- Cancel query (graceful)
SELECT pg_cancel_backend(12345);

-- Terminate connection (forceful)
SELECT pg_terminate_backend(12345);

-- Cancel all waiting > 5 min
SELECT pg_cancel_backend(pid)
FROM pg_stat_activity
WHERE wait_event_type = 'Lock'
  AND state_change < now() - interval '5 minutes';
```

Avoiding ACCESS EXCLUSIVE Locks

Keep your database available during schema changes

Why ACCESS EXCLUSIVE is Dangerous

Timeline of disaster:

1. ALTER TABLE needs ACCESS EXCLUSIVE
2. Waits behind long-running query
3. All new queries queue behind ALTER
4. Connection pool exhausted
5. Application timeout errors

Operations Taking ACCESS EXCLUSIVE

Operation	Lock	Alternative
DROP TABLE	ACCESS EXCLUSIVE	-
TRUNCATE	ACCESS EXCLUSIVE	DELETE in batches
VACUUM FULL / CLUSTER	ACCESS EXCLUSIVE	REPACK CONCURRENTLY (PG19?)
REINDEX	ACCESS EXCLUSIVE	REINDEX CONCURRENTLY (PG12+)
DROP INDEX	ACCESS EXCLUSIVE	DROP INDEX CONCURRENTLY
CREATE INDEX	SHARE (blocks writes)	CREATE INDEX CONCURRENTLY
REFRESH MAT VIEW	ACCESS EXCLUSIVE	REFRESH ... CONCURRENTLY
Most ALTER TABLE	ACCESS EXCLUSIVE	Various tricks...

CREATE INDEX CONCURRENTLY

- Takes longer (two table scans)
- Can fail and leave invalid index
- Cannot run inside a transaction
- Trade-offs:

```
-- BAD: Takes SHARE lock, blocks writes
CREATE INDEX idx_users_email ON users(email);

-- GOOD: Only SHARE UPDATE EXCLUSIVE
CREATE INDEX CONCURRENTLY idx_users_email ON users(email);
```

REINDEX CONCURRENTLY (PG 12+)

```
-- BAD: Blocks everything
REINDEX INDEX idx_users_email;

-- GOOD: Minimal locking
REINDEX INDEX CONCURRENTLY idx_users_email;

-- Reindex all indexes on a table
REINDEX TABLE CONCURRENTLY users;
```

DROP INDEX CONCURRENTLY

- Only one index at a time
- Cannot use CASCADE
- Cannot drop indexes backing constraints (PK, UNIQUE)
- Caveats:

```
-- BAD: ACCESS EXCLUSIVE on the table
DROP INDEX idx_users_email;

-- GOOD: No exclusive lock on table
DROP INDEX CONCURRENTLY idx_users_email;
```

Adding Columns Safely

```
-- PostgreSQL 11+: Adding column with DEFAULT is instant!
ALTER TABLE users ADD COLUMN created_at TIMESTAMP DEFAULT now();
-- Default stored in catalog, applied on read

-- Adding NOT NULL still needs care:
-- Step 1: Add column (instant)
ALTER TABLE users ADD COLUMN status TEXT DEFAULT 'active';

-- Step 2: Backfill if needed
-- Step 3: Add NOT NULL
ALTER TABLE users ALTER COLUMN status SET NOT NULL;
```

Adding Constraints Safely

```
-- BAD: Validates immediately, holds lock during scan
ALTER TABLE orders
ADD CONSTRAINT orders_user_fk
FOREIGN KEY (user_id) REFERENCES users(id);

-- GOOD: Two-step approach
ALTER TABLE orders
ADD CONSTRAINT orders_user_fk
FOREIGN KEY (user_id) REFERENCES users(id)
NOT VALID;

ALTER TABLE orders VALIDATE CONSTRAINT orders_user_fk;
```

Changing Column Types

```
-- BAD: Rewrites entire table with ACCESS EXCLUSIVE
ALTER TABLE users ALTER COLUMN age TYPE BIGINT;

-- GOOD: Create new column, migrate, swap
ALTER TABLE users ADD COLUMN age_new BIGINT;

-- Batch migration
UPDATE users SET age_new = age WHERE id BETWEEN 1 AND 10000;
-- Repeat...

-- Quick swap
ALTER TABLE users DROP COLUMN age,
                  RENAME COLUMN age_new TO age;
```

Best Practices Summary

- Keep transactions short and focused
- Avoid idle in transaction
- Lock resources in consistent order
- Use SKIP LOCKED for queue patterns
- Use CONCURRENTLY options for DDL
- Add constraints with NOT VALID , then VALIDATE
- Always SET lock_timeout before DDL
- Monitor pg_prepared_xacts if using 2PC

Thank you!

https://www.linkedin.com/in/pavandeolasee/