Pessimistic Locking

1. The Real Problem This Exists to Solve

When multiple transactions must modify the same data and conflicts are frequent or unacceptable, pessimistic locking prevents concurrent access by explicitly locking records before modification. This guarantees exclusive access and eliminates race conditions at the cost of concurrency.

Real production scenario:

- Banking system with account transfers
- Account A: $1000, Account B: $500
- **Without pessimistic locking:**
  - Transaction 1: Transfer $700 from A to B
  - Transaction 2: Transfer $600 from A to C (concurrent)
  - Both read A balance: $1000
  - Both check: $1000 >= $700 ✓ and $1000 >= $600 ✓
  - Both proceed
  - A ends with -$300 (overdraft!)
- **With pessimistic locking:**
  - Transaction 1: Lock account A, read $1000
  - Transaction 2: Tries to lock A, blocked/waits
  - Transaction 1: Transfer $700, A = $300, commit
  - Transaction 2: Now locks A, reads $300
  - Transaction 2: Check fails ($300 < $600), rollback
  - A ends with $300 (correct!)

The fundamental problem: Optimistic locking detects conflicts after they occur and requires retries. Pessimistic locking prevents conflicts by serializing access. When retries are expensive, dangerous, or conflicts are frequent, pessimistic locking is necessary.

Without pessimistic locking (high contention):

- Many retry attempts
- Wasted computation
- Starvation (some transactions never succeed)
- Race conditions on critical data

With pessimistic locking:

- Guaranteed exclusive access
- No retries needed
- Serialized execution
- Predictable behavior

2. The Naive / Incorrect Approaches (IMPORTANT)

❌ Incorrect Approach #1: Application-Level Locks (Not Database-Aware)

```
// Incorrect: In-memory locks don't coordinate with database transactions
const locks = new Map<string, boolean>();
```

```
async function transfer(fromId: number, toId: number, amount: number) {
  // Try to acquire lock
  if (locks.get(`account_${fromId}`)) {
    throw new Error('Account locked');
  }

  locks.set(`account_${fromId}`, true);

  try {
    // Begin transaction
    await db.query('BEGIN');

    const from = await db.query('SELECT balance FROM accounts WHERE id = $1',
[fromId]);

    if (from.rows[0].balance < amount) {
      throw new Error('Insufficient funds');
    }

    await db.query('UPDATE accounts SET balance = balance - $1 WHERE id = $2',
[amount, fromId]);
    await db.query('UPDATE accounts SET balance = balance + $1 WHERE id = $2',
[amount, toId]);

    await db.query('COMMIT');
  } finally {
    locks.delete(`account_${fromId}`);
  }
}
```

Why it seems reasonable:

- Simple Map-based locking
- Prevents concurrent access in application
- Easy to implement

How it breaks:

```
Server 1: locks.set('account_1', true) → proceeds
Server 2: locks.get('account_1') → undefined (different memory!)
Server 2: proceeds with same account
Both modify account_1 simultaneously
```

Production symptoms:

- Works on single server
- Breaks immediately when load balanced
- Race conditions in production (multi-instance)
- No coordination between application instances

❌ Incorrect Approach #2: Read Then Lock (Too Late)

```
// Incorrect: Check condition before locking
async function sellTicket(eventId: number, userId: number) {
  await db.query('BEGIN');

  // Read available tickets
  const event = await db.query(
    'SELECT available_tickets FROM events WHERE id = $1',
    [eventId]
  );

  if (event.rows[0].available_tickets <= 0) {
    await db.query('ROLLBACK');
    throw new Error('Sold out');
  }

  // Lock row (TOO LATE!)
  await db.query(
    'SELECT * FROM events WHERE id = $1 FOR UPDATE',
    [eventId]
  );

  // Update
  await db.query(
    'UPDATE events SET available_tickets = available_tickets - 1 WHERE id = $1',
    [eventId]
  );

  await db.query('COMMIT');
}
```

Why it seems reasonable:

- Checks availability first
- Locks before update
- Uses FOR UPDATE

How it breaks:

```
Time  | Transaction A                | Transaction B
------|------------------------------|------------------------------
T1    | Read: 1 ticket available     |
T2    |                              | Read: 1 ticket available
T3    | Lock row                     |
T4    |                              | Wait for lock...
T5    | Sell ticket (0 remaining)    |
T6    | COMMIT                       |
T7    |                              | Lock acquired
T8    |                              | Sell ticket (-1 remaining)
T9    |                              | COMMIT
```

Production symptoms:

- Overselling tickets/inventory
- Negative available count
- Check happened before lock
- Time-of-check vs time-of-use (TOCTOU) race

❌ Incorrect Approach #3: Locking Wrong Records

```
// Incorrect: Lock detail records but not parent
async function updateOrderTotal(orderId: number) {
  await db.query('BEGIN');

  // Lock order items
  const items = await db.query(
    'SELECT * FROM order_items WHERE order_id = $1 FOR UPDATE',
    [orderId]
  );

  const total = items.rows.reduce((sum, item) => sum + item.price * item.quantity,
0);

  // Update order total (parent not locked!)
  await db.query(
    'UPDATE orders SET total = $1 WHERE id = $2',
    [total, orderId]
  );

  await db.query('COMMIT');
}
```

Why it seems reasonable:

- Locks related records (order_items)
- Calculates total from locked items
- Updates parent record

How it breaks:

```
Transaction A: Locks order_items, calculates total = $100
Transaction B: Locks SAME order_items, calculates total = $100
Transaction A: Updates orders.total = $100
Transaction B: Updates orders.total = $100 (duplicate update)

OR:

Transaction A: Locks order_items
Transaction B: Reads orders (not locked), sees old total
Transaction A: Updates orders.total
Transaction B: Makes decision based on stale total
```

Production symptoms:

- Inconsistent totals

- Race conditions on parent record
- Lost updates on orders table
- Phantom reads of order totals

❌ Incorrect Approach #4: No Lock Timeout (Deadlock Prone)

```
// Incorrect: Lock without timeout, can deadlock
async function transferBetweenAccounts(from: number, to: number, amount: number) {
  await db.query('BEGIN');

  // Lock accounts in request order (not sorted!)
  const accountFrom = await db.query(
    'SELECT * FROM accounts WHERE id = $1 FOR UPDATE',
    [from]
  );

  const accountTo = await db.query(
    'SELECT * FROM accounts WHERE id = $1 FOR UPDATE',
    [to]
  );

  // Update balances...

  await db.query('COMMIT');
}
```

Why it seems reasonable:

- Locks both accounts
- Prevents concurrent modifications
- Looks safe

How it breaks (deadlock):

```
Transaction A: Lock account 1
Transaction B: Lock account 2
Transaction A: Try to lock account 2 (waits for B)
Transaction B: Try to lock account 1 (waits for A)
Deadlock! Both wait forever.
```

Production symptoms:

```
ERROR: deadlock detected
DETAIL: Process 12345 waits for ShareLock on transaction 67890;
        blocked by process 67891.
        Process 67891 waits for ShareLock on transaction 12346;
        blocked by process 12345.
```

3. Correct Mental Model (How It Actually Works)

Pessimistic locking acquires an exclusive lock on a database row **before** reading/modifying it. Other transactions trying to lock the same row will block and wait.

The SELECT FOR UPDATE Pattern

```sql
BEGIN;

-- Lock row for exclusive access
SELECT * FROM accounts WHERE id = 1 FOR UPDATE;

-- Now safe to modify
UPDATE accounts SET balance = balance - 100 WHERE id = 1;

COMMIT;
```

Lock Modes

```sql
-- Exclusive lock (blocks reads and writes)
SELECT * FROM table WHERE id = 1 FOR UPDATE;

-- Shared lock (allows reads, blocks writes)
SELECT * FROM table WHERE id = 1 FOR SHARE;

-- Skip locked rows (non-blocking variant)
SELECT * FROM table WHERE id = 1 FOR UPDATE SKIP LOCKED;

-- Wait with timeout
SET lock_timeout = '5s';
SELECT * FROM table WHERE id = 1 FOR UPDATE;
```

The Transaction Flow

```
1. BEGIN transaction
2. SELECT ... FOR UPDATE (acquire lock)
3. Check business rules
4. Perform updates
5. COMMIT (release lock)

If another transaction tries step 2 while lock held:
- It BLOCKS until lock released
- Or times out
- Or deadlock detected
```

Lock Ordering to Prevent Deadlock

```
Rule: Always lock resources in consistent order

Transfer from account A to B:
  Lock min(A, B) first
  Lock max(A, B) second
```

```
This prevents circular wait → no deadlock
```

4. Correct Design & Algorithm

Strategy 1: FOR UPDATE with Lock Ordering

```
-- Always lock accounts in ID order
BEGIN;

SELECT * FROM accounts WHERE id = LEAST($1, $2) FOR UPDATE;
SELECT * FROM accounts WHERE id = GREATEST($1, $2) FOR UPDATE;

-- Safe to modify both accounts
UPDATE accounts SET balance = balance - $amount WHERE id = $from;
UPDATE accounts SET balance = balance + $amount WHERE id = $to;

COMMIT;
```

Strategy 2: FOR UPDATE SKIP LOCKED (Queue Processing)

```
-- Workers grab next available job (non-blocking)
SELECT * FROM jobs
WHERE status = 'pending'
ORDER BY created_at
LIMIT 1
FOR UPDATE SKIP LOCKED;

-- Process job...
UPDATE jobs SET status = 'completed' WHERE id = ?;

COMMIT;
```

Strategy 3: Advisory Locks (Application-Level Coordination)

```
-- Named locks (PostgreSQL)
SELECT pg_advisory_lock(12345);
-- Do work
SELECT pg_advisory_unlock(12345);

-- Try lock (non-blocking)
SELECT pg_try_advisory_lock(12345);
```

5. Full Production-Grade Implementation

```
import { Pool, PoolClient } from 'pg';
```

```typescript
interface Account {
  id: number;
  balance: number;
  currency: string;
}

class DeadlockError extends Error {
  constructor(message: string) {
    super(message);
    this.name = 'DeadlockError';
  }
}

class InsufficientFundsError extends Error {
  constructor(
    message: string,
    public available: number,
    public requested: number
  ) {
    super(message);
    this.name = 'InsufficientFundsError';
  }
}

/**
 * Account service with pessimistic locking
 */
class AccountService {
  constructor(private pool: Pool) {}

  /**
   * Transfer money between accounts with pessimistic locking
   */
  async transfer(
    fromAccountId: number,
    toAccountId: number,
    amount: number
  ): Promise<void> {
    const client = await this.pool.connect();

    try {
      await client.query('BEGIN');

      // Set lock timeout to prevent infinite waits
      await client.query("SET lock_timeout = '10s'");

      // Lock accounts in sorted order to prevent deadlock
      const [firstId, secondId] = [fromAccountId, toAccountId].sort((a, b) => a -
b);

      const first = await this.lockAccount(client, firstId);
      const second = await this.lockAccount(client, secondId);
    // Determine which is from/to
    const fromAccount = first.id === fromAccountId ? first : second;
    const toAccount = first.id === toAccountId ? first : second;

    // Validate business rules
    if (fromAccount.balance < amount) {
      throw new InsufficientFundsError(
        `Insufficient funds in account ${fromAccountId}`,
        fromAccount.balance,
        amount
      );
    }

    if (fromAccount.currency !== toAccount.currency) {
      throw new Error('Currency mismatch');
    }

    // Perform transfer
    await client.query(
      'UPDATE accounts SET balance = balance - $1 WHERE id = $2',
      [amount, fromAccountId]
    );

    await client.query(
      'UPDATE accounts SET balance = balance + $1 WHERE id = $2',
      [amount, toAccountId]
    );

    // Record transaction
    await client.query(
      `INSERT INTO transactions (from_account, to_account, amount, timestamp)
       VALUES ($1, $2, $3, NOW())`,
      [fromAccountId, toAccountId, amount]
    );

    await client.query('COMMIT');
  } catch (error: any) {
    await client.query('ROLLBACK');

    // Detect deadlock
    if (error.code === '40P01') {
      throw new DeadlockError('Transfer failed due to deadlock, please retry');
    }

    // Lock timeout
    if (error.code === '55P03') {
      throw new Error('Transfer timed out waiting for lock');
    }

    throw error;
  } finally {
      client.release();
    }
  }

  /**
   * Lock account and return current state
   */
  private async lockAccount(client: PoolClient, accountId: number): Promise<Account>
{
    const result = await client.query(
      'SELECT id, balance, currency FROM accounts WHERE id = $1 FOR UPDATE',
      [accountId]
    );

    if (result.rows.length === 0) {
      throw new Error(`Account ${accountId} not found`);
    }

    return result.rows[0];
  }

  /**
   * Get account balance (no lock)
   */
  async getBalance(accountId: number): Promise<number> {
    const result = await this.pool.query(
      'SELECT balance FROM accounts WHERE id = $1',
      [accountId]
    );

    if (result.rows.length === 0) {
      throw new Error(`Account ${accountId} not found`);
    }

    return result.rows[0].balance;
  }

  /**
   * Lock account for read (shared lock, allows concurrent reads)
   */
  async getBalanceWithLock(accountId: number): Promise<number> {
    const client = await this.pool.connect();

    try {
      await client.query('BEGIN');

      const result = await client.query(
        'SELECT balance FROM accounts WHERE id = $1 FOR SHARE',
        [accountId]
      );

      await client.query('COMMIT');
      return result.rows[0].balance;
    } catch (error) {
      await client.query('ROLLBACK');
      throw error;
    } finally {
      client.release();
    }
  }
}

/**
 * Job queue with pessimistic locking
 */
interface Job {
  id: number;
  type: string;
  payload: any;
  status: 'pending' | 'processing' | 'completed' | 'failed';
  attempts: number;
  created_at: Date;
}

class JobQueue {
  constructor(private pool: Pool) {}

  /**
   * Claim next job (non-blocking with SKIP LOCKED)
   */
  async claimNextJob(): Promise<Job | null> {
    const client = await this.pool.connect();

    try {
      await client.query('BEGIN');

      // Grab first available job, skip if locked by another worker
      const result = await client.query<Job>(
        `SELECT * FROM jobs
         WHERE status = 'pending'
         ORDER BY created_at ASC
         LIMIT 1
         FOR UPDATE SKIP LOCKED`
      );

      if (result.rows.length === 0) {
        await client.query('ROLLBACK');
        return null;
      }

      const job = result.rows[0];

      // Mark as processing
      await client.query(
        `UPDATE jobs
         SET status = 'processing', attempts = attempts + 1
         WHERE id = $1`,
        [job.id]
      );

      await client.query('COMMIT');

      return { ...job, status: 'processing', attempts: job.attempts + 1 };
    } catch (error) {
      await client.query('ROLLBACK');
      throw error;
    } finally {
      client.release();
    }
  }

  /**
   * Complete job
   */
  async completeJob(jobId: number): Promise<void> {
    await this.pool.query(
      `UPDATE jobs SET status = 'completed', completed_at = NOW() WHERE id = $1`,
      [jobId]
    );
  }

  /**
   * Fail job
   */
  async failJob(jobId: number, error: string): Promise<void> {
    await this.pool.query(
      `UPDATE jobs SET status = 'failed', error = $1 WHERE id = $2`,
      [error, jobId]
    );
  }

  /**
   * Worker process
   */
  async processJobs(handler: (job: Job) => Promise<void>): Promise<void> {
    while (true) {
      const job = await this.claimNextJob();

      if (!job) {
        // No jobs available, wait
        await this.sleep(1000);
        continue;
      }

      try {
          await handler(job);
          await this.completeJob(job.id);
        } catch (error: any) {
          console.error(`Job ${job.id} failed:`, error);
          await this.failJob(job.id, error.message);
        }
      }
    }

  private sleep(ms: number): Promise<void> {
    return new Promise(resolve => setTimeout(resolve, ms));
  }
}

/**
 * Inventory management with pessimistic locking
 */
class InventoryService {
  constructor(private pool: Pool) {}

  /**
   * Reserve inventory (lock before check)
   */
  async reserveInventory(productId: number, quantity: number): Promise<void> {
    const client = await this.pool.connect();

    try {
      await client.query('BEGIN');
      await client.query("SET lock_timeout = '5s'");

      // Lock product row BEFORE checking availability
      const result = await client.query(
        'SELECT id, available FROM products WHERE id = $1 FOR UPDATE',
        [productId]
      );

      if (result.rows.length === 0) {
        throw new Error(`Product ${productId} not found`);
      }

      const product = result.rows[0];

      if (product.available < quantity) {
        throw new InsufficientFundsError(
          `Insufficient inventory for product ${productId}`,
          product.available,
          quantity
        );
      }

      // Deduct inventory
      await client.query(
        'UPDATE products SET available = available - $1 WHERE id = $2',
        [quantity, productId]
      );

      await client.query('COMMIT');
    } catch (error) {
      await client.query('ROLLBACK');
      throw error;
    } finally {
      client.release();
    }
  }
}

// Express API endpoints
const pool = new Pool({
  host: 'localhost',
  database: 'myapp',
  max: 20,
});

const accountService = new AccountService(pool);
const jobQueue = new JobQueue(pool);
const inventoryService = new InventoryService(pool);

/**
 * Transfer endpoint
 */
app.post('/api/transfer', async (req, res) => {
  try {
    const { fromAccountId, toAccountId, amount } = req.body;

    await accountService.transfer(fromAccountId, toAccountId, amount);

    res.json({ success: true, message: 'Transfer completed' });
  } catch (error: any) {
    if (error instanceof InsufficientFundsError) {
      res.status(400).json({
        error: error.message,
        available: error.available,
        requested: error.requested,
      });
    } else if (error instanceof DeadlockError) {
      res.status(409).json({ error: error.message });
    } else if (error.message.includes('timeout')) {
      res.status(408).json({ error: 'Request timeout' });
    } else {
      res.status(500).json({ error: 'Internal error' });
    }
  }
});
```

```
/**
 * Reserve inventory endpoint
 */
app.post('/api/inventory/reserve', async (req, res) => {
  try {
    const { productId, quantity } = req.body;

    await inventoryService.reserveInventory(productId, quantity);

    res.json({ success: true });
  } catch (error: any) {
    if (error instanceof InsufficientFundsError) {
      res.status(400).json({ error: error.message });
    } else {
      res.status(500).json({ error: 'Internal error' });
    }
  }
});

/**
 * Worker process for job queue
 */
async function startWorker() {
  console.log('Worker started');

  await jobQueue.processJobs(async (job) => {
    console.log(`Processing job ${job.id}: ${job.type}`);

    // Handle job based on type
    if (job.type === 'send_email') {
      await sendEmail(job.payload);
    } else if (job.type === 'generate_report') {
      await generateReport(job.payload);
    }
  });
}

// Start worker in separate process
if (process.env.WORKER === 'true') {
  startWorker();
}

/**
 * Example: Multiple workers processing jobs
 */
// Terminal 1: WORKER=true node app.js  (worker 1)
// Terminal 2: WORKER=true node app.js  (worker 2)
// Terminal 3: WORKER=true node app.js  (worker 3)
// All workers use SKIP LOCKED → no conflicts, each gets different job
```

6. Correct Usage Patterns (Where This Shines)

Pattern 1: Financial Transactions

```
await accountService.transfer(from, to, amount);
// Guaranteed no overdraft
// Serialized access to accounts
```

Pattern 2: Job Queue Processing

```
const job = await jobQueue.claimNextJob();
// Each worker gets different job
// No duplicate processing
```

Pattern 3: Ticket Sales / Limited Inventory

```
await inventoryService.reserveInventory(productId, quantity);
// No overselling
// Atomic check-and-reserve
```

7. Failure Modes & Edge Cases

Deadlock

Problem: Two transactions lock resources in opposite order.

Detection: Database detects cycle, aborts one transaction with error code 40P01.

Mitigation: Always lock in consistent order (e.g., by ID ascending).

Lock Timeout

Problem: Transaction waits too long for lock.

Detection: Error code 55P03 after timeout.

Mitigation: Set reasonable lock_timeout (5-10s), retry on timeout.

Lock Starvation

Problem: High-priority transaction keeps locking, low-priority never acquires.

Mitigation: Use queue with SKIP LOCKED, fair scheduling.

8. Performance Characteristics & Tradeoffs

Throughput

- **Pessimistic:** Lower (serial access)
- **Optimistic:** Higher (parallel access)

Latency

- **Pessimistic:** Predictable (no retries)
- **Optimistic:** Variable (retries on conflict)

Contention Handling

- **Pessimistic:** Blocks, waits in queue
- **Optimistic:** Immediate failure, retry

Best for High Contention

Pessimistic locking avoids wasted work from retries.

9. Foot-Guns & Common Mistakes (DO NOT SKIP)

Mistake 1: Locking After Reading

Problem: Check condition, then lock (TOCTOU race).

Fix: Lock FIRST with FOR UPDATE, then check.

Mistake 2: Wrong Lock Order

Problem: Deadlock from inconsistent lock ordering.

Fix: Always lock resources in sorted order (by ID).

Mistake 3: No Lock Timeout

Problem: Infinite wait if lock holder crashes.

Fix: SET lock_timeout = '10s' before locking.

Mistake 4: Holding Locks Too Long

Problem: Long-running transaction blocks others.

Fix: Keep transactions short, commit quickly.

Mistake 5: Using Application Locks Across Servers

Problem: In-memory locks don't coordinate between instances.

Fix: Use database locks (FOR UPDATE) or distributed locks (Redis).

10. When NOT to Use This (Anti-Patterns)

Low Contention

If conflicts are rare, optimistic locking is faster (no lock overhead).

Read-Heavy Workloads

Pessimistic locks block readers. Use MVCC or optimistic locking.

Long-Running Operations

Don't hold database locks during external API calls. Use queues instead.

11. Related Concepts (With Contrast)

Optimistic Locking

Difference: Pessimistic locks proactively, optimistic detects conflicts reactively.

Two-Phase Locking (2PL)

Related: Pessimistic locking is form of 2PL (acquire all locks, then release all).

Advisory Locks

Related: Application-defined locks using pg_advisory_lock().

Distributed Locks

Related: Pessimistic locking across services (Redis, etcd, ZooKeeper).

12. Production Readiness Checklist

Database Configuration

- [ ] Set lock_timeout (5-10s recommended)
- [ ] Set deadlock_timeout (1s default)
- [ ] Monitor lock waits in pg_stat_activity
- [ ] Enable log_lock_waits for debugging

Application Code

- [ ] Always BEGIN transaction before FOR UPDATE
- [ ] Lock resources in consistent order
- [ ] Handle deadlock errors (40P01) with retry
- [ ] Handle lock timeout errors (55P03)
- [ ] Keep transactions short (<100ms ideal)

Error Handling

- [ ] Detect deadlock (40P01) → retry
- [ ] Detect timeout (55P03) → return 408
- [ ] Log lock contention for analysis

Monitoring

- [ ] Dashboard: lock wait time
- [ ] Alert: deadlock rate >1/minute
- [ ] Track: transaction duration (p99)
- [ ] Identify: hot tables with high lock contention