Optimistic Locking

1. The Real Problem This Exists to Solve

When multiple users or processes concurrently update the same database record, later updates can overwrite earlier ones without knowing about them (lost update problem). Optimistic locking detects concurrent modifications and prevents silent data loss.

Real production scenario:

- E-commerce inventory management
- Current inventory: 10 units
- **Without optimistic locking:**
  - User A reads: inventory = 10
  - User B reads: inventory = 10
  - User A sells 3 units, writes: inventory = 7
  - User B sells 5 units, writes: inventory = 5
  - Final inventory: 5 (wrong! should be 2)
  - Lost Update: User A's sale of 3 units disappeared
- **With optimistic locking:**
  - User A reads: inventory = 10, version = 1
  - User B reads: inventory = 10, version = 1
  - User A updates: SET inventory = 7 WHERE id = X AND version = 1, SET version = 2
  - User B updates: SET inventory = 5 WHERE id = X AND version = 1
  - User B's update fails (version changed to 2)
  - User B retries: reads inventory = 7, version = 2
  - User B recalculates: 7 - 5 = 2, updates to 2
  - Final inventory: 2 (correct!)

The fundamental problem: Read-modify-write operations aren't atomic. Between reading data and writing updates, another process can modify the same data. Without detection, later updates clobber earlier ones, causing data loss and inconsistency.

Without optimistic locking:

- Lost updates (last write wins)
- No detection of concurrent modifications
- Silent data corruption
- Inventory discrepancies
- Financial errors

With optimistic locking:

- Detects concurrent modifications
- Rejects conflicting updates
- Forces retry with fresh data
- Prevents lost updates
- Maintains data integrity

2. The Naive / Incorrect Approaches (IMPORTANT)

❌ Incorrect Approach #1: Read-Modify-Write Without Version Check

```
// Incorrect: Classic lost update scenario
app.post('/api/inventory/sell', async (req, res) => {
  const { productId, quantity } = req.body;

  // Read current inventory
  const product = await db.query(
    'SELECT inventory FROM products WHERE id = $1',
    [productId]
  );

  const currentInventory = product.rows[0].inventory;

  // Calculate new inventory
  const newInventory = currentInventory - quantity;

  if (newInventory < 0) {
    return res.status(400).json({ error: 'Insufficient inventory' });
  }

  // Write new inventory (no version check!)
  await db.query(
    'UPDATE products SET inventory = $1 WHERE id = $2',
    [newInventory, productId]
  );

  res.json({ success: true, inventory: newInventory });
});
```

Why it seems reasonable:

- Simple logic: read → calculate → write
- Easy to understand
- No complex locking
- Works fine with single user

How it breaks:

```
Time  | Request A                    | Request B
------|------------------------------|------------------------------
T1    | Read: inventory = 10         |
T2    |                              | Read: inventory = 10
T3    | Calculate: 10 - 3 = 7        |
T4    |                              | Calculate: 10 - 5 = 5
T5    | Write: inventory = 7         |
T6    |                              | Write: inventory = 5
T7    | Final: inventory = 5 (WRONG: should be 2)
```

Production symptoms:

- Inventory shows 5 but should show 2
- Sold 8 units total (3 + 5) but only 5 deducted
- 3 units unaccounted for (inventory shrinkage)

- Monthly reconciliation shows discrepancies
- Lost revenue from free items

❌ Incorrect Approach #2: Timestamp-Based Checking (Race Condition)

```
// Incorrect: Check updated_at timestamp
app.post('/api/product/update', async (req, res) => {
  const { productId, newPrice, lastUpdated } = req.body;

  const product = await db.query(
    'SELECT * FROM products WHERE id = $1',
    [productId]
  );

  // Check if updated_at matches what client has
  if (product.rows[0].updated_at.getTime() !== new Date(lastUpdated).getTime()) {
    return res.status(409).json({ error: 'Product was modified by another user' });
  }

  // Update product
  await db.query(
    'UPDATE products SET price = $1, updated_at = NOW() WHERE id = $2',
    [newPrice, productId]
  );

  res.json({ success: true });
});
```

Why it seems reasonable:

- Detects if record changed
- Uses existing updated_at column
- No additional version column needed

How it breaks:

- Race condition between SELECT and UPDATE
- Two requests can pass the check simultaneously
- Timestamp precision issues (milliseconds)
- Clock skew between servers
- Still allows lost updates

Production symptoms:

```
T1: Request A checks updated_at (matches)
T2: Request B checks updated_at (matches)
T3: Request A updates (passes check)
T4: Request B updates (already passed check)
T5: Lost update still occurs
```

❌ Incorrect Approach #3: Application-Level Locking (Not Distributed Safe)

```
// Incorrect: In-memory lock (single server only)
const locks = new Map<number, boolean>();

app.post('/api/product/update', async (req, res) => {
  const { productId, newPrice } = req.body;

  // Check if locked
  if (locks.get(productId)) {
    return res.status(409).json({ error: 'Product is being edited' });
  }

  // Acquire lock
  locks.set(productId, true);

  try {
    const product = await db.query('SELECT * FROM products WHERE id = $1',
[productId]);
    await db.query('UPDATE products SET price = $1 WHERE id = $2', [newPrice,
productId]);
    res.json({ success: true });
  } finally {
    // Release lock
    locks.delete(productId);
  }
});
```

Why it seems reasonable:

- Prevents concurrent modifications
- Simple in-memory implementation
- Guarantees serial access

How it breaks:

- Only works on single server
- Multi-server deployment: each server has own locks (doesn't communicate)
- Lock not persisted (lost on restart)
- No lock timeout (deadlock if process crashes)
- Doesn't scale horizontally

Production symptoms:

- Works in development (single server)
- Fails in production (load balanced across 5 servers)
- Users on different servers can edit simultaneously
- No coordination between instances

❌ Incorrect Approach #4: SELECT FOR UPDATE Without Retry Logic

```
// Incorrect: Pessimistic lock but no retry on conflict
app.post('/api/product/update', async (req, res) => {
  const { productId, newPrice } = req.body;
```

```
  await db.query('BEGIN');

  try {
    // Lock row
    const product = await db.query(
      'SELECT * FROM products WHERE id = $1 FOR UPDATE',
      [productId]
    );

    // Update
    await db.query(
      'UPDATE products SET price = $1 WHERE id = $2',
      [newPrice, productId]
    );

    await db.query('COMMIT');
    res.json({ success: true });
  } catch (error) {
    await db.query('ROLLBACK');
    // No retry logic!
    res.status(500).json({ error: 'Update failed' });
  }
});
```

Why it seems reasonable:

- Uses database locking (correct)
- Prevents concurrent updates
- Atomic within transaction

How it breaks:

- Blocks other requests (not optimistic, it's pessimistic)
- High contention → many requests wait
- Deadlock potential
- No automatic retry
- User sees error instead of seamless retry

Production symptoms:

- High load → many requests blocked waiting for lock
- Timeout errors under heavy traffic
- Degraded performance (serialization)
- Lock wait timeouts

3. Correct Mental Model (How It Actually Works)

Optimistic locking assumes conflicts are rare. It adds a version field that increments on every update. Updates include a version check in the WHERE clause.

The Version Check Pattern

```
Table schema:
  products (
    id INT,
    name VARCHAR,
    price DECIMAL,
    inventory INT,
    version INT  ← Version field
  )

Read:
  SELECT id, name, price, inventory, version FROM products WHERE id = 1
  Result: { id: 1, name: "Widget", price: 10, inventory: 100, version: 5 }

Update with version check:
  UPDATE products
  SET inventory = 97, version = version + 1
  WHERE id = 1 AND version = 5

If UPDATE affects 0 rows → conflict (someone else updated)
If UPDATE affects 1 row → success
```

The Retry Loop

```
1. Read entity with version
2. Modify entity
3. Attempt update with version check
4. If update fails (0 rows affected):
   a. Re-read entity (get new version)
   b. Re-apply modifications with fresh data
   c. Retry update
5. If update succeeds → done
```

The Concurrent Update Flow

```
Initial state: inventory = 10, version = 1

Request A reads: inventory = 10, version = 1
Request B reads: inventory = 10, version = 1

Request A updates:
  UPDATE products SET inventory = 7, version = 2 WHERE id = X AND version = 1
  ✓ Success (1 row affected)

Request B updates:
  UPDATE products SET inventory = 5, version = 2 WHERE id = X AND version = 1
  ✗ Failed (0 rows affected, version is now 2 not 1)

Request B retries:
  Re-read: inventory = 7, version = 2
  Recalculate: 7 - 5 = 2
```

```
  UPDATE products SET inventory = 2, version = 3 WHERE id = X AND version = 2
  ✓ Success
```

Key Properties

1. No blocking: Readers don't block writers, writers don't block readers **2. Conflict detection:** Version mismatch detects concurrent modifications **3. Retry required:** Application must handle conflicts and retry **4. Performance:** Better than pessimistic locking under low contention

4. Correct Design & Algorithm

Strategy 1: Version Column

```sql
-- Add version column
ALTER TABLE products ADD COLUMN version INT DEFAULT 1;

-- Read with version
SELECT id, inventory, version FROM products WHERE id = ?;

-- Update with version check
UPDATE products
SET inventory = ?, version = version + 1
WHERE id = ? AND version = ?;

-- Check affected rows
IF affected_rows = 0 THEN
  RAISE conflict_error
END IF;
```

Strategy 2: Timestamp-Based (with atomic compare)

```sql
-- Use updated_at as version
UPDATE products
SET inventory = ?, updated_at = NOW()
WHERE id = ? AND updated_at = ?;
```

Strategy 3: CAS (Compare-And-Swap) Pattern

```
READ entity → get version V
MODIFY entity
UPDATE WHERE version = V
IF failed: retry from READ
```

5. Full Production-Grade Implementation

```typescript
interface Product {
  id: number;
  name: string;
```

```
  price: number;
  inventory: number;
  version: number;
}

class OptimisticLockError extends Error {
  constructor(
    message: string,
    public currentVersion: number,
    public attemptedVersion: number
  ) {
    super(message);
    this.name = 'OptimisticLockError';
  }
}

/**
 * Optimistic locking repository
 */
class ProductRepository {
  constructor(private db: Database) {}

  /**
   * Find product by ID with version
   */
  async findById(id: number): Promise<Product | null> {
    const result = await this.db.query(
      'SELECT id, name, price, inventory, version FROM products WHERE id = $1',
      [id]
    );
    return result.rows[0] || null;
  }

  /**
   * Update product with optimistic lock
   */
  async update(product: Product): Promise<Product> {
    const result = await this.db.query(
      `UPDATE products
       SET name = $1, price = $2, inventory = $3, version = version + 1
       WHERE id = $4 AND version = $5
       RETURNING id, name, price, inventory, version`,
      [product.name, product.price, product.inventory, product.id, product.version]
    );

    if (result.rowCount === 0) {
      // Conflict detected
      const current = await this.findById(product.id);
      throw new OptimisticLockError(
        `Product ${product.id} was modified by another transaction`,
        current?.version || -1,
        product.version
```

```
      );
    }

    return result.rows[0];
  }

  /**
   * Update with automatic retry
   */
  async updateWithRetry(
    id: number,
    updateFn: (product: Product) => Partial<Product>,
    maxRetries: number = 3
  ): Promise<Product> {
    let attempt = 0;
    let lastError: Error | null = null;

    while (attempt < maxRetries) {
      try {
        // Read current state
        const product = await this.findById(id);
        if (!product) {
          throw new Error(`Product ${id} not found`);
        }

        // Apply modifications
        const updates = updateFn(product);
        const updated = { ...product, ...updates };

        // Attempt update with version check
        return await this.update(updated);
      } catch (error) {
        if (error instanceof OptimisticLockError) {
          lastError = error;
          attempt++;

          // Exponential backoff
          const delay = Math.min(100 * Math.pow(2, attempt), 1000);
          await this.sleep(delay);

          continue;
        }
        throw error;
      }
    }

    throw new Error(
      `Failed to update product ${id} after ${maxRetries} retries. Last error:
${lastError?.message}`
    );
  }
```

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

/**
 * Service layer with business logic
 */
class InventoryService {
  constructor(private repository: ProductRepository) {}

  /**
   * Sell product with optimistic locking
   */
  async sellProduct(productId: number, quantity: number): Promise<Product> {
    return this.repository.updateWithRetry(
      productId,
      (product) => {
        // Check business rules
        if (product.inventory < quantity) {
          throw new Error(
            `Insufficient inventory. Available: ${product.inventory}, Requested:
${quantity}`
          );
        }

        // Calculate new inventory
        return {
          inventory: product.inventory - quantity,
        };
      },
      5  // Max 5 retries
    );
  }

  /**
   * Update price with optimistic locking
   */
  async updatePrice(productId: number, newPrice: number): Promise<Product> {
    return this.repository.updateWithRetry(
      productId,
      (product) => ({
        price: newPrice,
      })
    );
  }

  /**
   * Bulk update with optimistic locking
   */
  async bulkUpdateInventory(
    updates: Array<{ productId: number; quantitySold: number }>
```

```
  ): Promise<Product[]> {
    const results = await Promise.allSettled(
      updates.map(({ productId, quantitySold }) =>
        this.sellProduct(productId, quantitySold)
      )
    );

    const succeeded = results
      .filter((r): r is PromiseFulfilledResult<Product> => r.status === 'fulfilled')
      .map(r => r.value);

    const failed = results
      .filter((r): r is PromiseRejectedResult => r.status === 'rejected')
      .map(r => r.reason);

    if (failed.length > 0) {
      console.error(`Failed to update ${failed.length} products:`, failed);
    }

    return succeeded;
  }
}

// Express API endpoints
const productRepo = new ProductRepository(db);
const inventoryService = new InventoryService(productRepo);

/**
 * Sell product endpoint
 */
app.post('/api/products/:id/sell', async (req, res) => {
  try {
    const productId = parseInt(req.params.id);
    const { quantity } = req.body;

    const product = await inventoryService.sellProduct(productId, quantity);

    res.json({
      success: true,
      product,
      message: `Sold ${quantity} units. Remaining inventory: ${product.inventory}`,
    });
  } catch (error) {
    if (error.message.includes('Insufficient inventory')) {
      res.status(400).json({ error: error.message });
    } else if (error.message.includes('after') && error.message.includes('retries'))
{
      res.status(409).json({
        error: 'Unable to complete request due to high contention. Please try
again.',
      });
    } else {
```

```
      res.status(500).json({ error: 'Internal error' });
    }
  }
});

/**
 * Update price endpoint
 */
app.put('/api/products/:id/price', async (req, res) => {
  try {
    const productId = parseInt(req.params.id);
    const { price, version } = req.body;

    // Manual version check (alternative approach)
    const product = await productRepo.findById(productId);

    if (!product) {
      return res.status(404).json({ error: 'Product not found' });
    }

    if (product.version !== version) {
      return res.status(409).json({
        error: 'Product was modified by another user',
        currentVersion: product.version,
        yourVersion: version,
        currentData: product,
      });
    }

    product.price = price;
    const updated = await productRepo.update(product);

    res.json({ success: true, product: updated });
  } catch (error) {
    if (error instanceof OptimisticLockError) {
      res.status(409).json({
        error: error.message,
        currentVersion: error.currentVersion,
      });
    } else {
      res.status(500).json({ error: 'Internal error' });
    }
  }
});

/**
 * Client-side handling
 */
class ProductClient {
  async updatePrice(productId: number, newPrice: number): Promise<void> {
    const maxRetries = 3;
    let attempt = 0;
```

```
    while (attempt < maxRetries) {
      try {
        // Fetch current product with version
        const product = await this.fetchProduct(productId);

        // Update with version
        const response = await fetch(`/api/products/${productId}/price`, {
          method: 'PUT',
          headers: { 'Content-Type': 'application/json' },
          body: JSON.stringify({
            price: newPrice,
            version: product.version,
          }),
        });

        if (response.ok) {
          return; // Success
        }

        if (response.status === 409) {
          // Conflict, retry
          const data = await response.json();
          console.log('Conflict detected, retrying with fresh data:', data);
          attempt++;
          await this.sleep(100 * attempt);
          continue;
        }

        throw new Error(`HTTP ${response.status}`);
      } catch (error) {
        if (attempt === maxRetries - 1) {
          throw error;
        }
        attempt++;
      }
    }
  }

  private async fetchProduct(id: number): Promise<Product> {
    const response = await fetch(`/api/products/${id}`);
    return response.json();
  }

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

/**
 * React form with optimistic locking
 */
```

```tsx
function ProductEditForm({ productId }: { productId: number }) {
  const [product, setProduct] = useState<Product | null>(null);
  const [price, setPrice] = useState('');
  const [error, setError] = useState('');

  useEffect(() => {
    fetchProduct();
  }, [productId]);

  const fetchProduct = async () => {
    const response = await fetch(`/api/products/${productId}`);
    const data = await response.json();
    setProduct(data);
    setPrice(data.price.toString());
  };

  const handleSubmit = async (e: React.FormEvent) => {
    e.preventDefault();
    setError('');

    try {
      const response = await fetch(`/api/products/${productId}/price`, {
        method: 'PUT',
        headers: { 'Content-Type': 'application/json' },
        body: JSON.stringify({
          price: parseFloat(price),
          version: product!.version,
        }),
      });

      if (response.ok) {
        alert('Price updated successfully');
        fetchProduct(); // Refresh with new version
      } else if (response.status === 409) {
        const data = await response.json();
        setError(
          `This product was modified by another user. ` +
          `Current price: $${data.currentData.price}. ` +
          `Please refresh and try again.`
        );
        // Auto-refresh after showing error
        setTimeout(fetchProduct, 2000);
      }
    } catch (err) {
      setError('Failed to update price');
    }
  };

  return (
    <form onSubmit={handleSubmit}>
      <input
        type="number"
```

```
        value={price}
        onChange={(e) => setPrice(e.target.value)}
      />
      <button type="submit">Update Price</button>
      {error && <div className="error">{error}</div>}
      <small>Version: {product?.version}</small>
    </form>
  );
}
```

6. Correct Usage Patterns (Where This Shines)

Pattern 1: E-Commerce Inventory

```
await inventoryService.sellProduct(productId, quantity);
// Automatically retries on conflicts
// Prevents overselling
```

Pattern 2: Document Editing

```
await documentService.updateWithRetry(docId, (doc) => ({
  content: newContent,
  lastEditedBy: userId,
}));
// Multiple users can edit, conflicts auto-resolved
```

Pattern 3: Account Balance Updates

```
await accountService.updateWithRetry(accountId, (account) => ({
  balance: account.balance - amount,
}));
// Prevents lost updates on concurrent transactions
```

7. Failure Modes & Edge Cases

High Contention

Problem: Many concurrent updates → many retries → degraded performance.

Mitigation: Switch to pessimistic locking for hot records.

Retry Exhaustion

Problem: Max retries exceeded, update fails.

Mitigation: Increase max retries, add jitter, queue for later.

Version Overflow

Problem: Version column reaches max INT value.

Mitigation: Use BIGINT, or reset periodically.

8. Performance Characteristics & Tradeoffs

Low Contention

- **Optimistic:** Fast (no locking overhead)
- **Pessimistic:** Slower (lock overhead)
- **Winner:** Optimistic

High Contention

- **Optimistic:** Many retries, wasted work
- **Pessimistic:** Serialized, no retries
- **Winner:** Pessimistic

Throughput

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

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

Mistake 1: Forgetting to Check Affected Rows

Fix: Always check if UPDATE affected 1 row.

Mistake 2: No Retry Logic

Fix: Implement automatic retry with exponential backoff.

Mistake 3: Infinite Retry Loop

Fix: Set max retries (3-5), timeout after N attempts.

Mistake 4: Not Re-reading Data on Retry

Fix: Fetch fresh data before each retry attempt.

Mistake 5: Using optimistic lock on Hot Records

Fix: Switch to pessimistic locking for high-contention records.

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

High Contention Scenarios

If 100s of updates/second on same record, use pessimistic locking.

Financial Transactions

For money transfers, pessimistic locking may be safer (no retries).

Real-Time Collaboration

For Google Docs-style editing, use OT or CRDT instead.

11. Related Concepts (With Contrast)

Pessimistic Locking

Difference: Pessimistic locks immediately, blocks others. Optimistic detects conflicts later.

MVCC

Related: Multi-Version Concurrency Control uses similar versioning concept.

CAS Operations

Same concept: Compare-And-Swap is optimistic locking at CPU level.

12. Production Readiness Checklist

Database Schema

- [ ] Add version column (INT or BIGINT)
- [ ] Index on (id, version) for performance
- [ ] Set default value (version = 1)

Application Code

- [ ] Update queries include version check
- [ ] Check affected rows after UPDATE
- [ ] Implement retry logic (3-5 attempts)
- [ ] Exponential backoff between retries

Error Handling

- [ ] Return 409 Conflict on version mismatch
- [ ] Include current version in error response
- [ ] Client handles conflicts gracefully

Monitoring

- [ ] Track retry rate
- [ ] Alert on retry rate >10%
- [ ] Dashboard showing conflict frequency
- [ ] Identify hot records for optimization