Read-After-Write Consistency

1. The Real Problem This Exists to Solve

In distributed systems with read replicas, data written to the primary database takes time to replicate. If a user writes data and immediately reads it, they might see stale data if the read goes to a replica that hasn't received the update yet. Read-after-write consistency ensures users always see their own writes.

Real production scenario:

- Social media app with primary database + 5 read replicas
- User posts a new photo
- **Without read-after-write consistency:**
  - Write goes to primary: "Photo uploaded"
  - User redirected to profile page
  - Profile page reads from replica #3
  - Replica #3 hasn't replicated yet (50ms lag)
  - User sees: Photo not in their feed
  - User: "The upload failed! Let me try again..."
  - User uploads duplicate photo
  - 100ms later: Both photos appear (confused user)
- **With read-after-write consistency:**
  - Write goes to primary: "Photo uploaded"
  - User redirected to profile page
  - System detects: User is reading their own data
  - Read directed to primary (or waits for replication)
  - User sees: New photo immediately
  - User: "Perfect, upload succeeded!"

The fundamental problem: Replication lag is inevitable in distributed systems (typically 10-500ms). Users expect immediate consistency for their own writes even though eventual consistency is acceptable for other users' data. Without read-after-write consistency, users perceive the system as broken or slow.

Without read-after-write consistency:

- Users don't see their own changes
- Confusion and duplicate submissions
- Poor user experience
- Perceived as buggy
- Lost trust

With read-after-write consistency:

- Users always see their own writes
- Predictable behavior
- Better UX
- Additional complexity
- Slight performance overhead

2. The Naive / Incorrect Approaches (IMPORTANT)

❌ Incorrect Approach #1: Read From Random Replica (Eventual Consistency)

```typescript
// Incorrect: Load-balanced reads hit any replica
class PostService {
  async createPost(userId: number, content: string) {
    // Write to primary
    await primaryDb.query(
      'INSERT INTO posts (user_id, content) VALUES ($1, $2)',
      [userId, content]
    );

    return { success: true };
  }

  async getUserPosts(userId: number) {
    // Read from random replica (load balancer decides)
    const posts = await replicaDb.query(
      'SELECT * FROM posts WHERE user_id = $1 ORDER BY created_at DESC',
      [userId]
    );

    return posts.rows;
  }
}

// API flow
app.post('/api/posts', async (req, res) => {
  await postService.createPost(req.user.id, req.body.content);
  res.json({ success: true });
});

app.get('/api/posts', async (req, res) => {
  // User immediately fetches their posts
  const posts = await postService.getUserPosts(req.user.id);
  res.json(posts); // Might not include just-created post!
});
```

Why it seems reasonable:

- Writes to primary (correct)
- Reads from replicas (good for scale)
- Load balanced (even distribution)

How it breaks:

```
Time  | Action
------|--------------------------------------------------------
T0    | User creates post → writes to primary
T1    | Response: { success: true }
T2    | User's browser fetches posts → hits replica #2
T3    | Replica #2 is 100ms behind (replication lag)
T4    | Response: [...old posts] (new post missing!)
```

```
T5    | User: "WTF, it said success but post isn't there"
T100  | Replication completes, post now visible
```

Production symptoms:

- Users report "My post disappeared"
- Duplicate posts from retry attempts
- Support tickets: "Upload not working"
- Negative reviews: "App is buggy"

❌ Incorrect Approach #2: Always Read From Primary (Defeats Purpose of Replicas)

```
// Incorrect: Read everything from primary
class PostService {
  async getUserPosts(userId: number) {
    // Always read from primary (no replication lag)
    const posts = await primaryDb.query(
      'SELECT * FROM posts WHERE user_id = $1 ORDER BY created_at DESC',
      [userId]
    );

    return posts.rows;
  }
}
```

Why it seems reasonable:

- Solves read-after-write problem
- Always consistent
- Simple implementation

How it breaks:

- Primary database overloaded (reads + writes)
- Read replicas unused (wasted resources)
- Doesn't scale
- High latency from primary saturation

Production symptoms:

```
Primary DB CPU: 95%
Replica DB CPU: 10% (idle)
Read query latency: 500ms (should be 50ms)
Write query latency: 1000ms (queued behind reads)
```

❌ Incorrect Approach #3: Polling Until Data Appears

```
// Incorrect: Client polls until data visible
async function createPost(content: string) {
  await fetch('/api/posts', {
    method: 'POST',
    body: JSON.stringify({ content }),
```

```
  });

  // Poll until post appears
  let attempts = 0;
  while (attempts < 10) {
    const posts = await fetch('/api/posts').then(r => r.json());
    const newPost = posts.find(p => p.content === content);

    if (newPost) {
      return newPost; // Found it!
    }

    await sleep(100);
    attempts++;
  }

  throw new Error('Post not found after 1 second');
}
```

Why it seems reasonable:

- Eventually finds the data
- Handles replication lag
- Client-side solution

How it breaks:

- Wastes resources (10 requests vs 1)
- Slow (1 second delay)
- Still fails if lag > 1 second
- Race condition (multiple posts with same content)

Production symptoms:

```
Monitoring:
- Spike in /api/posts requests (10× normal)
- Network bandwidth wasted
- Client battery drain
- Slow page loads (perceived performance)
```

❌ Incorrect Approach #4: Fixed Delay After Write

```
// Incorrect: Wait fixed duration after write
async function createPost(content: string) {
  await fetch('/api/posts', {
    method: 'POST',
    body: JSON.stringify({ content }),
  });

  // Wait "long enough" for replication
  await sleep(200);
```

```
  // Now fetch
  const posts = await fetch('/api/posts').then(r => r.json());
  return posts;
}
```

Why it seems reasonable:

- Accounts for replication lag
- Simple to implement
- Usually works

How it breaks:

- Sometimes 200ms isn't enough (99th percentile lag: 500ms)
- Wastes time when lag is only 10ms
- Still no guarantee of consistency
- Slow user experience (forced wait)

Production symptoms:

```
P50 replication lag: 30ms (wasted 170ms)
P99 replication lag: 600ms (200ms not enough)
User sees stale data 1% of the time
Users complain about "sluggish" interface
```

❌ Incorrect Approach #5: Caching Write on Client (Stale on Refresh)

```
// Incorrect: Client-side cache hides the issue
const [posts, setPosts] = useState([]);

async function createPost(content: string) {
  const response = await fetch('/api/posts', {
    method: 'POST',
    body: JSON.stringify({ content }),
  });

  const newPost = await response.json();

  // Optimistically add to local state
  setPosts([newPost, ...posts]);
}

// On page refresh
useEffect(() => {
  // Fetch from server (might not have new post yet)
  fetch('/api/posts')
    .then(r => r.json())
    .then(setPosts);
}, []);
```

Why it seems reasonable:

- Instant feedback (optimistic UI)
- No waiting for server
- Feels fast

How it breaks:

- Refresh page → post disappears (replica lag)
- Share link → other users don't see post yet
- Client cache and server out of sync
- Breaks browser back button

Production symptoms:

```
User: Creates post
User: Sees post in UI (cached)
User: Refreshes page
User: Post gone!
User: Waits 2 seconds, refreshes again
User: Post appears
User: "This app is so buggy"
```

3. Correct Mental Model (How It Actually Works)

Read-after-write consistency ensures a user who writes data will read that data on subsequent requests. This is achieved by routing reads for that user to the primary database (or waiting for replication) for a short window after writes.

The Core Strategy

```
1. User writes data → primary database
2. System records: "User X wrote at timestamp T"
3. User reads data:
   - If read is for User X's own data AND read timestamp < T + grace period:
     → Read from primary
   - Else:
     → Read from replica (eventual consistency OK)
```

Session-Based Routing

```
// After write, store in session
session.lastWriteTime = Date.now();

// On read
if (isUserReadingOwnData && Date.now() - session.lastWriteTime < 5000) {
  // Read from primary (5-second grace period)
  return await primaryDb.query(...);
} else {
  // Read from replica
  return await replicaDb.query(...);
}
```

Replication Position Tracking

```sql
-- After write on primary
SELECT pg_current_wal_lsn(); -- Returns: '0/16B37D8'

-- On replica, check if caught up
SELECT pg_last_wal_replay_lsn(); -- Returns: '0/16B37D8'

-- If replica lsn >= primary lsn → safe to read
```

Session Stickiness

```
After write:
- Set cookie: primary_affinity=true, expires in 5s
- Subsequent reads check cookie
- If primary_affinity=true → route to primary
- After 5s, cookie expires → route to replica
```

4. Correct Design & Algorithm

Strategy 1: Session Stickiness (Read From Primary After Write)

```typescript
// Store last write time in session
interface Session {
  userId: number;
  lastWriteTime?: number;
}

function shouldReadFromPrimary(session: Session, resourceUserId: number): boolean {
  // Only for user's own data
  if (session.userId !== resourceUserId) {
    return false;
  }

  // Only within grace period (5 seconds)
  if (!session.lastWriteTime) {
    return false;
  }

  const elapsed = Date.now() - session.lastWriteTime;
  return elapsed < 5000;
}
```

Strategy 2: Replication Lag Tracking (Wait for Replica)

```typescript
// After write, get replication position
const primaryLSN = await primary.query('SELECT pg_current_wal_lsn()');
```

```
// Store in session
session.minReplicationLSN = primaryLSN;

// On read, wait for replica to catch up
async function waitForReplication(minLSN: string, timeout: number = 1000) {
  const start = Date.now();

  while (Date.now() - start < timeout) {
    const replicaLSN = await replica.query('SELECT pg_last_wal_replay_lsn()');

    if (replicaLSN >= minLSN) {
      return; // Caught up!
    }

    await sleep(10);
  }

  // Timeout, fallback to primary
  throw new Error('Replica lag timeout');
}
```

Strategy 3: Critical Read Flag (Explicit Primary Read)

```
// Mark critical reads
await postService.getUserPosts(userId, { criticalRead: true });

// Implementation
async getUserPosts(userId: number, options?: { criticalRead?: boolean }) {
  const db = options?.criticalRead ? primaryDb : replicaDb;
  return db.query('SELECT * FROM posts WHERE user_id = $1', [userId]);
}
```

5. Full Production-Grade Implementation

```
import { Pool } from 'pg';
import { Request, Response, NextFunction } from 'express';

/**
 * Database manager with primary/replica routing
 */
class DatabaseManager {
  constructor(
    private primary: Pool,
    private replicas: Pool[]
  ) {}

  /**
   * Execute write on primary
   */
```

```typescript
async write(query: string, params: any[]): Promise<any> {
  return this.primary.query(query, params);
}

/**
 * Execute read with routing logic
 */
async read(
  query: string,
  params: any[],
  options: {
    userId?: number;
    criticalRead?: boolean;
    session?: SessionData;
  } = {}
): Promise<any> {
  const shouldUsePrimary = this.shouldReadFromPrimary(options);

  if (shouldUsePrimary) {
    console.log('[DB] Reading from PRIMARY (read-after-write consistency)');
    return this.primary.query(query, params);
  }

  console.log('[DB] Reading from REPLICA');
  return this.getRandomReplica().query(query, params);
}

/**
 * Determine if should read from primary
 */
private shouldReadFromPrimary(options: {
  userId?: number;
  criticalRead?: boolean;
  session?: SessionData;
}): boolean {
  // Explicit critical read
  if (options.criticalRead) {
    return true;
  }

  // Check session for recent write
  if (options.session && options.userId) {
    const isOwnData = options.session.userId === options.userId;
    const recentWrite = this.hasRecentWrite(options.session);

    if (isOwnData && recentWrite) {
      return true;
    }
  }

  return false;
}
```

```typescript
  /**
   * Check if user wrote recently
   */
  private hasRecentWrite(session: SessionData): boolean {
    if (!session.lastWriteTime) {
      return false;
    }

    const elapsed = Date.now() - session.lastWriteTime;
    const gracePeriod = 5000; // 5 seconds

    return elapsed < gracePeriod;
  }

  /**
   * Get random replica for load balancing
   */
  private getRandomReplica(): Pool {
    return this.replicas[Math.floor(Math.random() * this.replicas.length)];
  }
}

/**
 * Session data
 */
interface SessionData {
  userId: number;
  lastWriteTime?: number;
}

/**
 * Express middleware to track writes
 */
function trackWrites(req: Request, res: Response, next: NextFunction) {
  const originalSend = res.json.bind(res);

  res.json = function (data: any) {
    // If this was a write operation, update session
    if (['POST', 'PUT', 'PATCH', 'DELETE'].includes(req.method)) {
      if (req.session) {
        req.session.lastWriteTime = Date.now();
      }
    }

    return originalSend(data);
  };

  next();
}

/**
```

```
 * Post service with read-after-write consistency
 */
class PostService {
  constructor(private db: DatabaseManager) {}

  /**
   * Create post (write to primary)
   */
  async createPost(userId: number, content: string): Promise<{ id: number }> {
    const result = await this.db.write(
      'INSERT INTO posts (user_id, content, created_at) VALUES ($1, $2, NOW())
RETURNING id',
      [userId, content]
    );

    return { id: result.rows[0].id };
  }

  /**
   * Get user's posts (read-after-write aware)
   */
  async getUserPosts(
    userId: number,
    session?: SessionData
  ): Promise<Array<{ id: number; content: string; created_at: Date }>> {
    const result = await this.db.read(
      'SELECT id, content, created_at FROM posts WHERE user_id = $1 ORDER BY
created_at DESC',
      [userId],
      { userId, session }
    );

    return result.rows;
  }

  /**
   * Get all posts (eventual consistency OK)
   */
  async getAllPosts(): Promise<Array<{ id: number; content: string }>> {
    const result = await this.db.read(
      'SELECT id, content FROM posts ORDER BY created_at DESC LIMIT 100',
      []
    );

    return result.rows;
  }

  /**
   * Get single post (critical read from primary)
   */
  async getPost(postId: number, options?: { criticalRead?: boolean }): Promise<any>
{
```

```typescript
    const result = await this.db.read(
      'SELECT * FROM posts WHERE id = $1',
      [postId],
      { criticalRead: options?.criticalRead }
    );

    return result.rows[0];
  }
}

/**
 * User profile service
 */
class ProfileService {
  constructor(private db: DatabaseManager) {}

  /**
   * Update profile (write)
   */
  async updateProfile(userId: number, name: string, bio: string): Promise<void> {
    await this.db.write(
      'UPDATE users SET name = $1, bio = $2 WHERE id = $3',
      [name, bio, userId]
    );
  }

  /**
   * Get profile (read-after-write aware)
   */
  async getProfile(userId: number, session?: SessionData): Promise<any> {
    const result = await this.db.read(
      'SELECT id, name, bio FROM users WHERE id = $1',
      [userId],
      { userId, session }
    );

    return result.rows[0];
  }
}

// Initialize
const primary = new Pool({ host: 'primary.db', database: 'myapp', max: 20 });
const replica1 = new Pool({ host: 'replica1.db', database: 'myapp', max: 50 });
const replica2 = new Pool({ host: 'replica2.db', database: 'myapp', max: 50 });

const dbManager = new DatabaseManager(primary, [replica1, replica2]);
const postService = new PostService(dbManager);
const profileService = new ProfileService(dbManager);

// Express app
app.use(trackWrites);
```

```
/**
 * Create post endpoint
 */
app.post('/api/posts', async (req, res) => {
  try {
    const { content } = req.body;
    const post = await postService.createPost(req.user.id, content);

    // Session now has lastWriteTime set by middleware
    res.json({ success: true, postId: post.id });
  } catch (error: any) {
    res.status(500).json({ error: error.message });
  }
});

/**
 * Get user's posts (read-after-write consistent)
 */
app.get('/api/posts', async (req, res) => {
  try {
    const posts = await postService.getUserPosts(
      req.user.id,
      req.session // Pass session for read-after-write logic
    );

    res.json(posts);
  } catch (error: any) {
    res.status(500).json({ error: error.message });
  }
});

/**
 * Get all posts (eventual consistency OK)
 */
app.get('/api/feed', async (req, res) => {
  try {
    const posts = await postService.getAllPosts();
    res.json(posts);
  } catch (error: any) {
    res.status(500).json({ error: error.message });
  }
});

/**
 * Update profile endpoint
 */
app.put('/api/profile', async (req, res) => {
  try {
    const { name, bio } = req.body;
    await profileService.updateProfile(req.user.id, name, bio);

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

```
  } catch (error: any) {
    res.status(500).json({ error: error.message });
  }
});

/**
 * Get profile (read-after-write consistent)
 */
app.get('/api/profile/:userId', async (req, res) => {
  try {
    const userId = parseInt(req.params.userId);
    const profile = await profileService.getProfile(userId, req.session);

    res.json(profile);
  } catch (error: any) {
    res.status(500).json({ error: error.message });
  }
});

/**
 * Alternative: Cookie-based approach
 */
function shouldReadFromPrimaryCookie(req: Request, userId: number): boolean {
  const primaryAffinity = req.cookies.primary_affinity;

  if (!primaryAffinity) {
    return false;
  }

  // Check if cookie is for this user
  return primaryAffinity === userId.toString();
}

app.post('/api/posts-v2', async (req, res) => {
  const { content } = req.body;
  await postService.createPost(req.user.id, content);

  // Set cookie for 5 seconds
  res.cookie('primary_affinity', req.user.id.toString(), {
    maxAge: 5000,
    httpOnly: true,
  });

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

/**
 * Advanced: Replication position tracking
 */
class ReplicationAwareDB extends DatabaseManager {
  async writeWithPosition(query: string, params: any[]): Promise<{ result: any; lsn:
string }> {
```

```
    const result = await this.primary.query(query, params);

    // Get current LSN (Log Sequence Number)
    const lsnResult = await this.primary.query('SELECT pg_current_wal_lsn()');
    const lsn = lsnResult.rows[0].pg_current_wal_lsn;

    return { result, lsn };
  }

  async readFromReplicaWhenReady(
    query: string,
    params: any[],
    minLSN: string,
    timeout: number = 1000
  ): Promise<any> {
    const replica = this.getRandomReplica();
    const start = Date.now();

    while (Date.now() - start < timeout) {
      // Check replica position
      const lsnResult = await replica.query('SELECT pg_last_wal_replay_lsn()');
      const replicaLSN = lsnResult.rows[0].pg_last_wal_replay_lsn;

      if (replicaLSN >= minLSN) {
        // Replica caught up, safe to read
        return replica.query(query, params);
      }

      // Wait 10ms and retry
      await new Promise(resolve => setTimeout(resolve, 10));
    }

    // Timeout, fallback to primary
    console.warn('[DB] Replica lag timeout, reading from primary');
    return this.primary.query(query, params);
  }
}
```

6. Correct Usage Patterns (Where This Shines)

Pattern 1: User Profile Updates

```
// User updates bio
await profileService.updateProfile(userId, name, bio);

// Immediately view profile → reads from primary
const profile = await profileService.getProfile(userId, session);
```

Pattern 2: Post Creation

```
// Create post
await postService.createPost(userId, content);

// Fetch posts → includes new post (primary read)
const posts = await postService.getUserPosts(userId, session);
```

Pattern 3: Shopping Cart

```
// Add item to cart
await cartService.addItem(userId, productId);

// View cart → sees new item (primary read)
const cart = await cartService.getCart(userId, session);
```

7. Failure Modes & Edge Cases

Replica Lag Spike

Problem: Replication lag exceeds grace period (5s), user sees stale data.

Mitigation: Increase grace period to 10s, monitor lag, alert if >1s.

Session Lost

Problem: Session expires, user no longer routes to primary.

Mitigation: Use longer session TTL, or LSN-based tracking.

Primary Overload

Problem: Too many reads to primary after writes.

Mitigation: Optimize grace period, use replica wait strategy.

8. Performance Characteristics & Tradeoffs

Read Latency

- **With read-after-write:** Slight increase (primary may be farther, more loaded)
- **Without:** Lower (replicas closer, less loaded)

Primary Load

- **With read-after-write:** 5-10% more reads on primary
- **Without:** All reads on replicas

Consistency

- **With read-after-write:** Strong for own writes
- **Without:** Eventual (see stale data)

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

Mistake 1: Forgetting to Pass Session

Fix: Middleware automatically adds session to request context.

Mistake 2: Grace Period Too Short

Fix: Monitor P99 replication lag, set grace period to P99 + buffer.

Mistake 3: Routing All Reads to Primary

Fix: Only route user's own data, not other users' data.

Mistake 4: Not Handling Primary Failover

Fix: If primary fails, promote replica, update routing logic.

Mistake 5: Session Stored in Memory (Lost on Restart)

Fix: Use Redis or database-backed sessions.

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

Single Database

If no replicas, read-after-write is automatic (not needed).

Analytics Queries

Don't need read-after-write for batch analytics (eventual consistency OK).

Immutable Event Logs

Append-only logs don't need read-after-write (never modified).

11. Related Concepts (With Contrast)

Eventual Consistency

Difference: Read-after-write is stronger than eventual (guarantees own writes visible).

Strong Consistency

Difference: Strong consistency guarantees all writes visible to all users (read-after-write only for own writes).

Session Affinity

Related: Sticky sessions route user to same server (similar concept).

Causal Consistency

Related: Guarantees causally related operations visible in order.

12. Production Readiness Checklist

Database Setup

- [ ] Primary database + N read replicas
- [ ] Monitor replication lag (CloudWatch, Datadog)
- [ ] Alert if replication lag >500ms
- [ ] Test failover (primary → replica promotion)

Application Code

- [ ] Middleware tracks writes in session
- [ ] Read logic checks session for recent writes
- [ ] Grace period set to P99 lag + 2× buffer
- [ ] Fallback to primary if replica unavailable

Session Management

- [ ] Use Redis or DB-backed sessions (not memory)
- [ ] Session TTL > grace period
- [ ] Handle session expiration gracefully

Monitoring

- [ ] Track % of reads from primary vs replica
- [ ] Alert if primary read % >15%
- [ ] Dashboard: replication lag by replica
- [ ] Log stale read incidents