Read Replicas & Replica Lag

1. The Real Problem This Exists to Solve

Databases have limited read capacity. As traffic grows, a single primary database becomes a bottleneck. Read replicas distribute read traffic across multiple servers, but replication isn't instant. Replica lag (the delay between primary writes and replica updates) causes stale reads, inconsistent data, and application bugs if not handled properly.

Real production scenario:

- E-commerce platform with 10,000 reads/second
- Single primary database saturated at 5,000 reads/second
- **Without read replicas:**
  - Primary CPU: 100%
  - Query latency: 2000ms (queue buildup)
  - Writes also slow (same server)
  - Frequent timeouts
  - Service degradation
- **With read replicas but no lag handling:**
  - Deploy 5 read replicas
  - Primary handles writes + critical reads
  - Replicas handle bulk reads
  - Average replica lag: 50ms
  - User posts review → writes to primary
  - User refreshes page → reads from replica (lag: 100ms)
  - Review doesn't appear (replica hasn't caught up)
  - User: "My review disappeared!"
  - Bug reports flood in
  - Trust issues
- **With read replicas and proper lag handling:**
  - User posts review → writes to primary
  - Store write timestamp in session
  - User refreshes → check if user reading own data
  - Route to primary for user's own recent writes
  - Route to replica for other users' data
  - User sees review immediately
  - Other users see it after 50-100ms (acceptable)
  - Zero complaints

The fundamental problem: Replication is asynchronous and eventual. The primary accepts writes immediately, but replicas lag behind by milliseconds to seconds. Applications must account for this delay or face data consistency issues, especially for read-after-write scenarios where users expect to see their own changes immediately.

Without replica awareness:

- Stale data visible to users
- Read-after-write inconsistency
- Confusing user experience
- Data appears to "disappear"

- Lost trust

With proper replica handling:

- Read-after-write consistency maintained
- Horizontal read scalability
- Predictable behavior
- Happy users
- 10× read capacity

2. The Naive / Incorrect Approaches (IMPORTANT)

❌ Incorrect Approach #1: Random Replica Selection (Ignoring Lag)

```
// Incorrect: Load balance reads randomly across replicas
const replicas = [replica1, replica2, replica3, replica4, replica5];

function getRandomReplica() {
  return replicas[Math.floor(Math.random() * replicas.length)];
}

app.get('/api/posts/:id', async (req, res) => {
  const db = getRandomReplica();

  const post = await db.query('SELECT * FROM posts WHERE id = $1', [req.params.id]);

  res.json(post);
});

app.post('/api/posts', async (req, res) => {
  // Write to primary
  await primary.query('INSERT INTO posts (content) VALUES ($1)',
[req.body.content]);

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

Why it seems reasonable:

- Distributes load evenly
- Simple round-robin or random selection
- All replicas get traffic

How it breaks:

```
Timeline:
T0: User creates post → writes to primary (id=123)
T1: Replication lag on replicas:
    - replica1: 30ms behind
    - replica2: 50ms behind
    - replica3: 100ms behind
    - replica4: 20ms behind
    - replica5: 200ms behind
```

```
T2: User clicks to view post (id=123)
T3: Random selection picks replica3 (100ms behind)
T4: Query: SELECT * FROM posts WHERE id = 123
T5: Result: 0 rows (replica hasn't received data yet)
T6: User sees: "Post not found"
T7: User: "WTF, I just created it!"

5 seconds later:
T5000: User refreshes
T5001: Random selection picks replica4 (caught up now)
T5002: Post appears
T5003: User: "This app is so buggy"
```

Production symptoms:

```
Support tickets:
- "My data disappeared"
- "Posted comment, but it's not showing"
- "Uploaded photo, can't see it"

Monitoring:
- Replica lag: P50=30ms, P99=500ms
- User complaints: 5% of write operations
- Pattern: Always after writes, fixed by refresh
```

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

```
// Incorrect: Avoid lag by reading everything from primary
app.get('/api/posts/:id', async (req, res) => {
  // Always use primary to avoid stale data
  const post = await primary.query('SELECT * FROM posts WHERE id = $1',
[req.params.id]);

  res.json(post);
});

app.get('/api/feed', async (req, res) => {
  // Also read from primary
  const posts = await primary.query('SELECT * FROM posts ORDER BY created_at DESC
LIMIT 100');

  res.json(posts);
});
```

Why it seems reasonable:

- No stale data
- Always consistent
- No replica lag issues

How it breaks:

```
Primary database:
- Writes: 1,000/second
- Reads: 10,000/second (should go to replicas)
- Total: 11,000/second
- Capacity: 5,000/second
- Result: OVERLOADED

Metrics:
- Primary CPU: 100%
- Query latency: 2000ms
- Read replicas: 0% CPU (idle, wasted resources)
- Failed requests: 50%
- Service outage
```

Production symptoms:

```
Database monitoring:
- Primary: 100% CPU, 5000 active connections
- Replica1: 5% CPU, 0 connections (unused)
- Replica2: 5% CPU, 0 connections (unused)
- Replica3: 5% CPU, 0 connections (unused)

Cost:
- Primary: $10,000/month (massive instance)
- 3 Replicas: $6,000/month (idle)
- Total wasted: $6,000/month
```

❌ Incorrect Approach #3: Fixed Wait After Write

```
// Incorrect: Wait for replication after every write
app.post('/api/posts', async (req, res) => {
  await primary.query('INSERT INTO posts (content) VALUES ($1)',
[req.body.content]);

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

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

app.get('/api/posts/:id', async (req, res) => {
  // Now safe to read from replica (maybe...)
  const db = getRandomReplica();
  const post = await db.query('SELECT * FROM posts WHERE id = $1', [req.params.id]);

  res.json(post);
});
```

Why it seems reasonable:

- Gives time for replication

- Simple implementation
- Usually works

How it breaks:

```
Scenario 1: Lag spike
- Typical lag: 30ms
- Wait time: 200ms
- Works 99% of time

- Replica lag spike: 500ms (database backup running)
- Wait time: 200ms (not enough)
- Still returns stale data
- User sees 404

Scenario 2: Unnecessary delay
- Replica lag: 10ms (fast)
- Wait time: 200ms
- Wasted 190ms
- Slow API response
- Poor user experience

Performance impact:
- Every write artificially delayed 200ms
- Throughput: 5 writes/second per user (vs 50/second)
- Users experience "sluggish" app
```

Production symptoms:

```
Monitoring:
- P99 write latency: 250ms (200ms artificial delay)
- P50 replica lag: 30ms (wasted 170ms per write)
- Throughput: 10× slower than necessary
- User feedback: "App feels slow"
```

❌ Incorrect Approach #4: Sticky Sessions to Same Replica (Hot Spots)

```
// Incorrect: Pin user to specific replica forever
const userReplicaMap = new Map<string, Database>();

app.use((req, res, next) => {
  const userId = req.user?.id;

  if (!userId) {
    return next();
  }

  // Assign user to replica once
  if (!userReplicaMap.has(userId)) {
    const replica = replicas[userId % replicas.length];
    userReplicaMap.set(userId, replica);
```

```
  }

  req.replicaDb = userReplicaMap.get(userId);
  next();
});

app.get('/api/posts', async (req, res) => {
  const posts = await req.replicaDb.query('SELECT * FROM posts LIMIT 100');
  res.json(posts);
});
```

Why it seems reasonable:

- User always reads from same replica
- Consistent view of data
- Session affinity

How it breaks:

```
User distribution (not uniform):
- 1000 users
- User IDs: 1-1000
- 5 replicas

Distribution:
- replica1 (id % 5 = 0): 200 users
- replica2 (id % 5 = 1): 200 users
- replica3 (id % 5 = 2): 200 users
- replica4 (id % 5 = 3): 200 users
- replica5 (id % 5 = 4): 200 users

But actual user activity (power users):
- User 42: 1,000 requests/second (power user, bot?)
- User 42 % 5 = 2 → replica3
- Replica3: OVERLOADED
- Replica1,2,4,5: Idle

Hot spot:
- Replica3: 100% CPU
- Other replicas: 20% CPU
- Uneven load distribution
- Slow queries for users on replica3
```

Production symptoms:

```
Metrics:
- Replica1: 500 req/s, 30% CPU
- Replica2: 400 req/s, 25% CPU
- Replica3: 5000 req/s, 100% CPU (hot spot!)
- Replica4: 600 req/s, 35% CPU
- Replica5: 500 req/s, 30% CPU
```

```
Users on replica3: Slow queries, timeouts
Users on other replicas: Fast queries
```

❌ Incorrect Approach #5: Polling Primary Until Data Appears

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

  // Poll until post appears on replica
  const maxAttempts = 20;
  for (let i = 0; i < maxAttempts; i++) {
    const posts = await fetch('/api/posts').then(r => r.json());

    if (posts.find(p => p.content === content)) {
      return; // Found it!
    }

    await sleep(100);
  }

  throw new Error('Post not replicated after 2 seconds');
}
```

Why it seems reasonable:

- Waits for replication to complete
- Eventually consistent
- Handles lag gracefully

How it breaks:

```
Performance:
- 1 write → 20 additional read requests (polling)
- Network: 20× overhead
- Database: 20× read load
- Battery drain on mobile

Scalability:
- 1,000 writes/second
- Each triggers 10 polls (average)
- Total read requests: 10,000/second from polling alone
- Original reads: 5,000/second
- Total: 15,000 reads/second
- Replicas overloaded by polling

Cost:
- API calls: 15× more than necessary
```

```
- Database reads: 3× more than necessary
- Infrastructure cost: +200%
```

3. Correct Mental Model (How It Actually Works)

Read replicas asynchronously replicate data from the primary. Replication lag is the time difference between a write on the primary and when it appears on replicas. Applications must route reads intelligently based on consistency requirements.

Replication Flow

```
Primary DB:
  Write arrives at T0
  → Committed to WAL (Write-Ahead Log)
  → Data written to disk
  → Replication stream sends WAL to replicas

Replica1:
  Receives WAL at T0 + 20ms
  → Applies changes
  → Data visible at T0 + 30ms
  → Lag: 30ms

Replica2:
  Receives WAL at T0 + 50ms (network delay)
  → Applies changes
  → Data visible at T0 + 60ms
  → Lag: 60ms
```

Consistency Levels

```
Strong consistency: Read from primary (no lag)
  - Use case: User's own data after write

Eventual consistency: Read from replica (acceptable lag)
  - Use case: Feed, list of all posts

Bounded staleness: Read from replica with max lag
  - Use case: Read from replica if lag < 100ms, else primary
```

Routing Strategy

```
Decision tree:
1. Is this user reading their own recent write?
   YES → Route to primary
   NO → Continue

2. Is strong consistency required?
   YES → Route to primary
   NO → Continue
```

```
3. Is replica lag acceptable?
   YES → Route to replica
   NO → Route to primary or wait for replica
```

4. Correct Design & Algorithm

Strategy 1: Session-Based Primary Routing

```
// Track recent writes in session
session.lastWriteTime = Date.now();

// Route reads
if (isUserOwnData && Date.now() - session.lastWriteTime < 5000) {
  return primary; // Read-after-write consistency
} else {
  return replica; // Eventual consistency OK
}
```

Strategy 2: LSN-Based Replication Tracking

```
// After write, get LSN
const lsn = await primary.query('SELECT pg_current_wal_lsn()');

// Wait for replica to catch up
while (await replica.query('SELECT pg_last_wal_replay_lsn()') < lsn) {
  await sleep(10);
}

// Now safe to read from replica
```

Strategy 3: Lag Monitoring

```
// Monitor lag
const lag = await replica.query(`
  SELECT
    now() - pg_last_xact_replay_timestamp() AS lag
  FROM pg_stat_replication
`);

if (lag > 1000) {
  // Lag too high, route to primary
  return primary;
}
```

5. Full Production-Grade Implementation

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

/**
 * Database cluster with primary and replicas
 */
class DatabaseCluster {
  private primary: Pool;
  private replicas: Pool[];
  private replicaLag: Map<Pool, number> = new Map();

  constructor(
    primaryConfig: any,
    replicaConfigs: any[]
  ) {
    this.primary = new Pool(primaryConfig);
    this.replicas = replicaConfigs.map(config => new Pool(config));

    this.startLagMonitoring();
  }

  /**
   * Monitor replica lag
   */
  private startLagMonitoring() {
    setInterval(async () => {
      for (const replica of this.replicas) {
        try {
          const result = await replica.query(`
            SELECT EXTRACT(EPOCH FROM (now() - pg_last_xact_replay_timestamp())) *
1000 AS lag_ms
          `);

          const lag = result.rows[0]?.lag_ms || 0;
          this.replicaLag.set(replica, lag);

          if (lag > 1000) {
            console.warn(`[Replica] High lag detected: ${lag.toFixed(0)}ms`);
          }
        } catch (error) {
          console.error('[Replica] Failed to check lag:', error);
          this.replicaLag.set(replica, Infinity); // Mark as unavailable
        }
      }
    }, 5000); // Check every 5 seconds
  }

  /**
   * Get primary database (for writes and strong consistency reads)
   */
  getPrimary(): Pool {
    return this.primary;
```

```
  }

  /**
   * Get best replica (lowest lag, round-robin among healthy)
   */
  getReplica(options: { maxLag?: number } = {}): Pool {
    const maxLag = options.maxLag || 1000; // Default: 1 second max

    // Filter replicas by lag threshold
    const healthyReplicas = this.replicas.filter(replica => {
      const lag = this.replicaLag.get(replica) || 0;
      return lag < maxLag;
    });

    if (healthyReplicas.length === 0) {
      console.warn('[Replica] No healthy replicas, falling back to primary');
      return this.primary;
    }

    // Round-robin among healthy replicas
    return healthyReplicas[Math.floor(Math.random() * healthyReplicas.length)];
  }

  /**
   * Get replica lag stats
   */
  getLagStats() {
    const lags = Array.from(this.replicaLag.values());

    return {
      min: Math.min(...lags),
      max: Math.max(...lags),
      avg: lags.reduce((a, b) => a + b, 0) / lags.length,
      replicas: Array.from(this.replicaLag.entries()).map(([replica, lag]) => ({
        lag,
        healthy: lag < 1000,
      })),
    };
  }

  /**
   * Wait for replica to catch up to specific LSN
   */
  async waitForReplication(replica: Pool, minLSN: string, timeout: number = 1000):
Promise<boolean> {
    const start = Date.now();

    while (Date.now() - start < timeout) {
      const result = await replica.query('SELECT pg_last_wal_replay_lsn()');
      const currentLSN = result.rows[0].pg_last_wal_replay_lsn;

      if (currentLSN >= minLSN) {
```

```
        return true; // Caught up
      }

      await new Promise(resolve => setTimeout(resolve, 10));
    }

    return false; // Timeout
  }
}

/**
 * Initialize cluster
 */
const dbCluster = new DatabaseCluster(
  {
    host: 'primary.db.example.com',
    database: 'myapp',
    max: 20,
  },
  [
    { host: 'replica1.db.example.com', database: 'myapp', max: 50 },
    { host: 'replica2.db.example.com', database: 'myapp', max: 50 },
    { host: 'replica3.db.example.com', database: 'myapp', max: 50 },
  ]
);

/**
 * Session tracking for read-after-write
 */
interface UserSession {
  userId: string;
  lastWriteTime?: number;
  lastWriteLSN?: string;
}

/**
 * Middleware: Track writes
 */
app.use((req, res, next) => {
  const originalJson = res.json.bind(res);

  res.json = function (data: any) {
    if (['POST', 'PUT', 'PATCH', 'DELETE'].includes(req.method)) {
      // Mark that user performed a write
      if (req.session) {
        req.session.lastWriteTime = Date.now();
      }
    }

    return originalJson(data);
  };
```

```
  next();
});

/**
 * Smart read routing
 */
class QueryRouter {
  constructor(private cluster: DatabaseCluster) {}

  /**
   * Route read query based on consistency requirements
   */
  getReadDatabase(options: {
    userId?: string;
    session?: UserSession;
    strongConsistency?: boolean;
    maxLag?: number;
  }): Pool {
    // Strong consistency required → primary
    if (options.strongConsistency) {
      return this.cluster.getPrimary();
    }

    // Read-after-write: User reading their own recent write
    if (options.session && options.userId) {
      const isOwnData = options.session.userId === options.userId;
      const recentWrite = this.hasRecentWrite(options.session);

      if (isOwnData && recentWrite) {
        console.log('[Router] Read-after-write detected, routing to primary');
        return this.cluster.getPrimary();
      }
    }

    // Eventual consistency OK → replica
    return this.cluster.getReplica({ maxLag: options.maxLag });
  }

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

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

    return elapsed < gracePeriod;
  }
```

```
  /**
   * Get write database (always primary)
   */
  getWriteDatabase(): Pool {
    return this.cluster.getPrimary();
  }
}

const router = new QueryRouter(dbCluster);

/**
 * Example: List posts (eventual consistency OK)
 */
app.get('/api/posts', async (req, res) => {
  try {
    const db = router.getReadDatabase({
      maxLag: 500, // Max 500ms lag acceptable
    });

    const result = await db.query('SELECT * FROM posts ORDER BY created_at DESC
LIMIT 100');

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

/**
 * Example: Get user's own posts (read-after-write consistency)
 */
app.get('/api/users/:userId/posts', async (req, res) => {
  try {
    const userId = req.params.userId;

    const db = router.getReadDatabase({
      userId,
      session: req.session,
      // Will route to primary if user wrote recently
    });

    const result = await db.query(
      'SELECT * FROM posts WHERE user_id = $1 ORDER BY created_at DESC',
      [userId]
    );

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

```
/**
 * Example: Create post (write to primary)
 */
app.post('/api/posts', async (req, res) => {
  try {
    const db = router.getWriteDatabase();

    const result = await db.query(
      'INSERT INTO posts (user_id, content) VALUES ($1, $2) RETURNING *',
      [req.user.id, req.body.content]
    );

    // Session middleware will mark lastWriteTime automatically

    res.json(result.rows[0]);
  } catch (error: any) {
    res.status(500).json({ error: error.message });
  }
});

/**
 * Example: Critical read (strong consistency required)
 */
app.get('/api/account/balance', async (req, res) => {
  try {
    const db = router.getReadDatabase({
      strongConsistency: true, // Always read from primary
    });

    const result = await db.query(
      'SELECT balance FROM accounts WHERE user_id = $1',
      [req.user.id]
    );

    res.json({ balance: result.rows[0].balance });
  } catch (error: any) {
    res.status(500).json({ error: error.message });
  }
});

/**
 * Monitoring endpoint
 */
app.get('/metrics/replication', (req, res) => {
  const stats = dbCluster.getLagStats();

  res.json({
    replicationLag: {
      min: stats.min,
      max: stats.max,
      avg: stats.avg,
    },
```

```
    replicas: stats.replicas,
  });
});

/**
 * Advanced: LSN-based consistency
 */
class LSNRouter extends QueryRouter {
  async readWithConsistency(
    query: string,
    params: any[],
    options: { minLSN?: string; timeout?: number } = {}
  ): Promise<any> {
    if (!options.minLSN) {
      // No LSN requirement, use normal routing
      const db = this.getReadDatabase({});
      return db.query(query, params);
    }

    // Try replica first
    const replica = this.cluster.getReplica();
    const caughtUp = await this.cluster.waitForReplication(
      replica,
      options.minLSN,
      options.timeout || 1000
    );

    if (caughtUp) {
      console.log('[LSN] Replica caught up, reading from replica');
      return replica.query(query, params);
    }

    // Replica too slow, fallback to primary
    console.log('[LSN] Replica lag exceeded timeout, reading from primary');
    return this.cluster.getPrimary().query(query, params);
  }
}

/**
 * Health check with replication status
 */
app.get('/health', (req, res) => {
  const stats = dbCluster.getLagStats();

  // Unhealthy if max lag > 5 seconds
  if (stats.max > 5000) {
    return res.status(503).json({
      status: 'unhealthy',
      reason: 'High replication lag',
      maxLag: stats.max,
    });
  }
```

```
  // Unhealthy if no healthy replicas
  const healthyCount = stats.replicas.filter(r => r.healthy).length;
  if (healthyCount === 0) {
    return res.status(503).json({
      status: 'unhealthy',
      reason: 'No healthy replicas',
    });
  }

  res.status(200).json({
    status: 'healthy',
    replicationLag: stats,
    healthyReplicas: healthyCount,
  });
});
```

6. Correct Usage Patterns (Where This Shines)

Pattern 1: Feed/List (Eventual Consistency)

```
// 100ms lag acceptable for public feed
const db = router.getReadDatabase({ maxLag: 100 });
const posts = await db.query('SELECT * FROM posts LIMIT 100');
```

Pattern 2: Own Data (Read-After-Write)

```
// User viewing their own profile
const db = router.getReadDatabase({ userId, session });
// Routes to primary if recent write
```

Pattern 3: Financial Data (Strong Consistency)

```
// Account balance must be accurate
const db = router.getReadDatabase({ strongConsistency: true });
const balance = await db.query('SELECT balance FROM accounts WHERE id = $1', [id]);
```

7. Failure Modes & Edge Cases

Replication Lag Spike

Problem: Replica falls behind (backup, network issue).

Mitigation: Monitor lag, route to primary if lag >1s.

Replica Failure

Problem: Replica crashes or becomes unavailable.

Mitigation: Health checks, automatic failover to primary.

Split Brain

Problem: Network partition, replica thinks it's primary.

Mitigation: Use connection strings, don't promote replicas manually.

8. Performance Characteristics & Tradeoffs

Read Scalability

- **Primary only:** 5,000 reads/second
- **With 5 replicas:** 25,000 reads/second (5× improvement)

Consistency

- **Primary:** Strong consistency, 0ms lag
- **Replica:** Eventual consistency, 30-500ms lag

Cost

- **Primary:** $1,000/month (large instance)
- **5 Replicas:** $500/month each = $2,500/month
- **Total:** $3,500/month (supports 5× traffic)

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

Mistake 1: Random Replica Without Lag Check

Fix: Monitor lag, route to primary if lag >1s.

Mistake 2: Always Reading From Primary

Fix: Route user's own data to primary, everything else to replica.

Mistake 3: No Session Tracking

Fix: Track writes in session, route subsequent reads to primary.

Mistake 4: Ignoring Lag Spikes

Fix: Alert when lag >1s, investigate root cause.

Mistake 5: Sticky Sessions Creating Hot Spots

Fix: Use round-robin among healthy replicas, not user affinity.

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

Write-Heavy Workloads

If 80% writes, 20% reads, replicas don't help much.

Strong Consistency Requirements

If all data must be immediately consistent, use primary only.

Small Scale

If <1,000 queries/second, single database sufficient.

11. Related Concepts (With Contrast)

Read-After-Write Consistency

Related: Handled by routing user's recent writes to primary.

Multi-Master Replication

Difference: Read replicas are read-only, multi-master allows writes to any node.

Sharding

Difference: Replicas copy all data, shards split data across nodes.

12. Production Readiness Checklist

Replication Setup

- [ ] Configure primary → replica replication
- [ ] Enable WAL archiving
- [ ] Set replication slots (prevent WAL deletion)
- [ ] Monitor replication connections

Application Code

- [ ] Implement read routing (primary vs replica)
- [ ] Track user writes in session
- [ ] Route read-after-write to primary
- [ ] Fallback to primary if replicas unhealthy

Monitoring

- [ ] Track replication lag per replica
- [ ] Alert when lag >1 second
- [ ] Dashboard: lag over time
- [ ] Monitor replica health

Performance

- [ ] Load test with replicas
- [ ] Verify 5-10× read capacity increase
- [ ] Measure P99 lag (should be <500ms)
- [ ] Test failover (replica → primary)