Cache Invalidation Strategies

1. The Real Problem This Exists to Solve

Caches improve performance by storing frequently accessed data in fast storage (memory) instead of repeatedly querying slow storage (database, API). But caches become stale when underlying data changes. Serving stale data causes bugs, shows outdated information to users, and breaks business logic.

Real production scenario:

- E-commerce product page cached in Redis
- Product price: $100 (cached)
- Admin updates price to $80 (database updated)
- **Without cache invalidation:**
  - User views product: Sees $100 (stale cache)
  - User adds to cart: Price $100
  - User checks out: Charged $100
  - Actual price should be $80
  - Customer complaint: "I was overcharged $20!"
  - Refund required, customer trust lost
  - Cache remains stale until expiration (possibly hours)
  - More customers overcharged
- **With cache invalidation:**
  - Admin updates price to $80 (database updated)
  - Cache invalidation triggered: DELETE cache key "product:123"
  - User views product: Cache miss, fetches from DB ($80)
  - User sees correct price immediately
  - No overcharges, no refunds, no complaints
  - Cache rebuilt with fresh data

The fundamental problem: Caches and databases store duplicate data. When database changes, cache doesn't automatically update. This creates a consistency problem: which version is correct? Without invalidation, caches serve increasingly stale data, breaking application correctness.

Two hard problems in computer science:

1. Naming things
2. **Cache invalidation** ← This one
3. Off-by-one errors

Without cache invalidation:

- Stale data served to users
- Users see outdated information
- Business logic breaks (pricing errors)
- Manual cache clearing required
- Inconsistent state across services

With cache invalidation:

- Fresh data after updates
- Immediate consistency
- Correct business logic

- Automated cache management
- Predictable behavior

2. The Naive / Incorrect Approaches (IMPORTANT)

❌ Incorrect Approach #1: Only TTL, No Invalidation (Stale for Hours)

```
// Incorrect: Rely solely on TTL, no active invalidation
async function getProduct(id: string) {
  const cached = await redis.get(`product:${id}`);
  if (cached) return JSON.parse(cached);

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

  // Cache for 1 hour
  await redis.setex(`product:${id}`, 3600, JSON.stringify(product));

  return product;
}

// Update product (no invalidation!)
async function updateProduct(id: string, updates: any) {
  await db.query('UPDATE products SET price = $1 WHERE id = $2', [updates.price,
id]);
  // Cache NOT invalidated, remains stale for up to 1 hour
}
```

Why it seems reasonable:

- Simple caching implementation
- TTL prevents infinite staleness
- No complex invalidation logic needed

How it breaks:

```
Timeline:
10:00 - Product cached: { id: 1, price: 100 }
10:05 - Admin updates price to 80 (database)
10:05 - Cache still: { id: 1, price: 100 }
10:30 - User views product: $100 (wrong!)
10:45 - User purchases: charged $100
11:00 - Cache expires
11:01 - Next request: Cache miss, fetches $80 (correct)

Problem:
- 55 minutes of stale data
- All users saw wrong price
- Multiple overcharges
- Customer complaints

"Why didn't you just clear the cache when updating?"
"We rely on TTL... didn't think about invalidation."
```

```
Worst case:
- Critical data (pricing, inventory, permissions)
- Long TTL (24 hours)
- Data changes frequently
- Users see wrong data for entire day
```

Production symptoms:

```
Customer reports:
"Price on website says $100, but cart shows $80?"
"Inventory shows 'In Stock' but checkout says 'Out of Stock'"
"I bought premium membership but still seeing ads"

Database:
price = 80 (correct)

Cache:
price = 100 (stale, will expire in 40 minutes)

Support tickets:
- 50 complaints about pricing discrepancies
- 20 complaints about inventory errors
- Manual cache flush required
```

❌ Incorrect Approach #2: Global Cache Flush (Nuclear Option)

```
// Incorrect: Clear entire cache on any update
async function updateProduct(id: string, updates: any) {
  await db.query('UPDATE products SET price = $1 WHERE id = $2', [updates.price,
id]);

  // Nuclear option: Flush entire cache!
  await redis.flushall();
}
```

Why it seems reasonable:

- Guarantees no stale data
- Simple to implement
- No need to track keys

How it breaks:

```
Cache state before update:
- product:1 (accessed 1M times/day)
- product:2 (accessed 500K times/day)
- product:3 (accessed 800K times/day)
- ...10,000 other cached products

Update product:1:
await redis.flushall(); // Clear EVERYTHING!
```

```
Immediate impact:
- All 10,000 products uncached
- Next requests: 10,000 cache misses
- Database: Receives 10,000 queries simultaneously
- Database CPU: 10% → 90%
- Database connections: 20 → 200 (exhausted)
- Query time: 50ms → 5000ms (timeout)
- Application: 500 errors
- Users: "Website is down"

"We updated one product and broke the entire site."
```

Production symptoms:

```
Monitoring:
12:00:00 - Update product:1
12:00:01 - redis.flushall() executed
12:00:02 - Cache hit rate: 80% → 0%
12:00:03 - Database queries: 100/sec → 10,000/sec
12:00:05 - Database CPU: 90%
12:00:07 - Application timeouts: 95%
12:00:10 - Service degraded

Incident:
Duration: 20 minutes (until cache warmed up)
Impact: 90% of requests failed
Root cause: Global cache flush
Resolution: Don't flush entire cache, only invalidate specific keys
```

❌ Incorrect Approach #3: No Invalidation on Indirect Changes (Forgotten Dependencies)

```
// Incorrect: Invalidate product cache, forget category cache
async function updateProduct(id: string, updates: any) {
  await db.query('UPDATE products SET name = $1 WHERE id = $2', [updates.name, id]);

  // Invalidate product cache
  await redis.del(`product:${id}`);

  // BUG: Forgot to invalidate category cache!
  // Category page still shows old product name
}

// Category page (cached)
async function getCategoryProducts(categoryId: string) {
  const cached = await redis.get(`category:${categoryId}:products`);
  if (cached) return JSON.parse(cached);

  const products = await db.query(
    'SELECT * FROM products WHERE category_id = $1',
```

```
    [categoryId]
  );

  await redis.setex(`category:${categoryId}:products`, 3600,
JSON.stringify(products));
  return products;
}
```

Why it seems reasonable:

- Invalidated the direct cache (product)
- Didn't realize category cache depends on product data
- Hard to track all dependencies

How it breaks:

```
Data relationships:
product:123 → "Blue Widget"
category:5:products → [{ id: 123, name: "Blue Widget" }, ...]

Update:
1. Update product name: "Blue Widget" → "Red Widget"
2. Invalidate product:123 cache
3. Forget to invalidate category:5:products

Result:
- Product page: Shows "Red Widget" (correct, cache invalidated)
- Category page: Shows "Blue Widget" (stale, cache not invalidated)
- Inconsistent data across pages

User experience:
- User browses category: Sees "Blue Widget"
- User clicks product: Sees "Red Widget"
- User confused: "Did the product change while I was browsing?"

Worse: Cascading dependencies
- Products depend on categories
- Categories depend on brands
- Brands depend on manufacturers
- Change manufacturer name → must invalidate:
  * manufacturer cache
  * brand cache
  * category cache
  * product cache
- Forgot one? Stale data remains.
```

Production symptoms:

```
Bug report:
"Product search shows old product name, but product page shows new name."

Investigation:
```

```
- product:123 cache: Invalidated ✓
- search:results:widget cache: NOT invalidated ✗
- category:5:products cache: NOT invalidated ✗
- recommendations:user:456 cache: NOT invalidated ✗

Root cause:
- Product data duplicated in 4 different caches
- Only 1 cache invalidated
- 3 caches still stale
- Developer didn't realize dependencies
```

❌ Incorrect Approach #4: Time-Based Invalidation for Critical Data (Race Condition)

```
// Incorrect: Use stale-while-revalidate for user permissions
async function getUserPermissions(userId: string) {
  const cached = await redis.get(`permissions:${userId}`);

  if (cached) {
    // Return stale data immediately
    const permissions = JSON.parse(cached);

    // Revalidate in background (not awaited!)
    this.revalidatePermissions(userId); // Fire and forget

    return permissions;
  }

  // Cache miss: Fetch from DB
  return this.fetchAndCachePermissions(userId);
}

// Background revalidation
async function revalidatePermissions(userId: string) {
  const fresh = await db.query('SELECT * FROM permissions WHERE user_id = $1',
[userId]);
  await redis.setex(`permissions:${userId}`, 60, JSON.stringify(fresh));
}
```

Why it seems reasonable:

- Fast response time (always return cached)
- Background revalidation keeps cache fresh
- Works well for non-critical data

How it breaks:

```
Security scenario:
10:00 - User has "admin" permission (cached)
10:05 - Admin revokes user's permission (database updated)
10:06 - User makes request
10:06 - Cache returns: "admin" (STALE! Still has access)
```

```
10:06 - Background revalidation started (not awaited)
10:07 - User accesses admin panel (should be denied)
10:07 - Check permissions: "admin" (from stale cache)
10:07 - Access granted ✓ (WRONG!)
10:08 - Background revalidation completes
10:08 - Cache updated: No admin permission

Result:
- User accessed admin panel for 2 minutes after revocation
- Security breach
- Unauthorized access logged

Worse scenario:
- Fire employee at 5pm
- Revoke access immediately
- Stale-while-revalidate keeps permissions cached
- Ex-employee accesses system for 60 seconds (cache TTL)
- Downloads confidential data
- Security incident
```

Production symptoms:

```
Audit log:
17:00:00 - User 'john' permissions revoked
17:00:30 - User 'john' accessed admin panel (how?!)
17:01:00 - User 'john' downloaded customer database
17:02:00 - Cache revalidated, access denied

Security team:
"How did revoked user access admin panel?"
"Stale cache with 60s TTL."
"WHY ARE PERMISSIONS CACHED WITH STALE-WHILE-REVALIDATE?!"

Lesson:
Never use stale-while-revalidate for:
- Permissions/authorization
- Pricing
- Inventory (can oversell)
- Financial data
- Medical records
```

❌ Incorrect Approach #5: Write-Through Cache Without Error Handling (Data Loss)

```
// Incorrect: Write to cache and DB, but no transaction/rollback
async function updateProduct(id: string, updates: any) {
  // Write to database
  await db.query('UPDATE products SET price = $1 WHERE id = $2', [updates.price,
id]);

  // Write to cache
```

```
  await redis.set(`product:${id}`, JSON.stringify(updates)); // What if this fails?
}
```

Why it seems reasonable:

- Keep cache and database in sync
- No separate invalidation step needed
- Fast reads (always cached)

How it breaks:

```
Scenario 1: Redis connection fails
await db.query(...); // ✓ Success (database updated)
await redis.set(...); // ✗ Error (Redis down)

Result:
- Database: price = 80 (updated)
- Cache: price = 100 (not updated, error thrown)
- Next request: Returns cached $100 (stale)
- Database and cache out of sync

Scenario 2: Redis slow, timeout
await db.query(...); // ✓ Success (50ms)
await redis.set(...); // ✗ Timeout (Redis overloaded, 5000ms)

Result:
- Request times out
- Client retries
- Database updated twice (race condition)
- Cache may or may not update
- Inconsistent state

Scenario 3: Partial failure
await db.query(...); // ✓ Success
await redis.set(`product:${id}`, ...); // ✓ Success
// Application crashes before response sent
// Client retries
// Database updated again, cache stale
```

Production symptoms:

```
Errors:
Error: Redis connection timeout
  at updateProduct (product.service.ts:45)

Monitoring:
- Database writes: 100% success
- Cache writes: 20% failure rate
- Cache/DB consistency: 80%

User impact:
- 20% of updates not reflected immediately
```

```
- Stale data served
- Price discrepancies
- Inventory errors

Incident:
"Redis had 5 minute outage.
All product updates during outage:
- Saved to database ✓
- NOT saved to cache ✗
Cache now 100% stale.
Manual flush required."
```

3. Correct Mental Model (How It Actually Works)

Cache invalidation strategies:

1. Write-Through (Update cache and DB)

```
Client writes data
  ↓
Update database
  ↓
Update cache
  ↓
Return success
```

Pros: Cache always fresh **Cons:** Slow writes (2 operations), cache can fail

2. Cache-Aside (Invalidate cache, lazy load)

```
Client writes data
  ↓
Update database
  ↓
Delete cache key
  ↓
Next read: Cache miss → Load from DB → Cache result
```

Pros: Simple, resilient, cache rebuild on demand **Cons:** First read after write is slow (cache miss)

3. Write-Behind (Async write to DB)

```
Client writes data
  ↓
Update cache immediately
  ↓
Return success
  ↓
Async: Update database
```

Pros: Fast writes **Cons:** Risk of data loss if cache fails before DB write

4. Event-Driven Invalidation

```
Write to database
  ↓
Emit event: "product.updated"
  ↓
Event handler: Invalidate cache keys
  ↓
Cache rebuilt on next read
```

Pros: Decoupled, handles complex dependencies **Cons:** Eventual consistency, requires event infrastructure

4. Correct Design & Algorithm

Cache-Aside (Most Common)

```
READ:
1. Check cache
2. If hit: Return cached data
3. If miss: Query database
4. Store result in cache
5. Return data

WRITE:
1. Update database
2. Delete cache key
3. Next read will rebuild cache
```

Write-Through

```
WRITE:
1. Update database (in transaction)
2. If success: Update cache
3. If cache fails: Log error, invalidate key
4. Return success
```

Dependency Tracking

```
Product → Categories, Search, Recommendations

When product updated:
1. Invalidate product:${id}
2. Invalidate category:${categoryId}:products
3. Invalidate search:*
4. Invalidate recommendations:*
```

5. Full Production-Grade Implementation

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

/**
 * Cache-aside pattern with automatic invalidation
 */
class CacheManager {
  private dependencyMap: Map<string, Set<string>> = new Map();

  constructor(
    private redis: Redis,
    private db: Pool
  ) {
    this.setupDependencies();
  }

  /**
   * Define cache dependencies
   */
  private setupDependencies() {
    // When product changes, invalidate these caches
    this.dependencyMap.set('product', new Set([
      'category:*:products',
      'search:*',
      'recommendations:*',
    ]));

    // When category changes, invalidate these caches
    this.dependencyMap.set('category', new Set([
      'category:*',
      'products:by-category:*',
    ]));
  }

  /**
   * Get with cache-aside pattern
   */
  async get<T>(
    key: string,
    fetcher: () => Promise<T>,
    ttl: number = 3600
  ): Promise<T> {
    try {
      // Check cache
      const cached = await this.redis.get(key);
      if (cached) {
        console.log(`[Cache] HIT: ${key}`);
        return JSON.parse(cached);
      }

      console.log(`[Cache] MISS: ${key}`);
```

```
    // Cache miss: Fetch from source
    const data = await fetcher();

    // Store in cache
    await this.redis.setex(key, ttl, JSON.stringify(data));

    return data;

  } catch (error) {
    console.error(`[Cache] Error for key ${key}:`, error);
    // Fallback: Return data without cache
    return fetcher();
  }
}

/**
 * Invalidate single key
 */
async invalidate(key: string): Promise<void> {
  try {
    await this.redis.del(key);
    console.log(`[Cache] Invalidated: ${key}`);
  } catch (error) {
    console.error(`[Cache] Failed to invalidate ${key}:`, error);
  }
}

/**
 * Invalidate multiple keys by pattern
 */
async invalidatePattern(pattern: string): Promise<void> {
  try {
    const keys = await this.redis.keys(pattern);

    if (keys.length === 0) {
      console.log(`[Cache] No keys match pattern: ${pattern}`);
      return;
    }

    await this.redis.del(...keys);
    console.log(`[Cache] Invalidated ${keys.length} keys matching: ${pattern}`);

  } catch (error) {
    console.error(`[Cache] Failed to invalidate pattern ${pattern}:`, error);
  }
}

/**
 * Invalidate with dependencies
 */
async invalidateWithDependencies(entityType: string, entityId: string):
```

```
Promise<void> {
    const primaryKey = `${entityType}:${entityId}`;

    // Invalidate primary key
    await this.invalidate(primaryKey);

    // Invalidate dependent keys
    const dependencies = this.dependencyMap.get(entityType) || new Set();

    for (const pattern of dependencies) {
      await this.invalidatePattern(pattern);
    }

    console.log(`[Cache] Invalidated ${entityType}:${entityId} with dependencies`);
  }

  /**
   * Set value in cache
   */
  async set(key: string, value: any, ttl: number = 3600): Promise<void> {
    try {
      await this.redis.setex(key, ttl, JSON.stringify(value));
    } catch (error) {
      console.error(`[Cache] Failed to set ${key}:`, error);
    }
  }
}

/**
 * Product repository with cache invalidation
 */
class ProductRepository {
  constructor(
    private db: Pool,
    private cache: CacheManager
  ) {}

  /**
   * Get product (cache-aside)
   */
  async getProduct(id: string): Promise<any> {
    return this.cache.get(
      `product:${id}`,
      async () => {
        const result = await this.db.query(
          'SELECT * FROM products WHERE id = $1',
          [id]
        );
        return result.rows[0];
      },
      3600 // TTL: 1 hour
    );
```

```
  }

  /**
   * Update product (invalidate cache)
   */
  async updateProduct(id: string, updates: any): Promise<void> {
    const client = await this.db.connect();

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

      // Update database
      await client.query(
        'UPDATE products SET name = $1, price = $2 WHERE id = $3',
        [updates.name, updates.price, id]
      );

      await client.query('COMMIT');

      // Invalidate cache with dependencies
      await this.cache.invalidateWithDependencies('product', id);

      console.log(`[Product] Updated product ${id}`);

    } catch (error) {
      await client.query('ROLLBACK');
      console.error(`[Product] Failed to update ${id}:`, error);
      throw error;
    } finally {
      client.release();
    }
  }

  /**
   * Get category products (with cache)
   */
  async getCategoryProducts(categoryId: string): Promise<any[]> {
    return this.cache.get(
      `category:${categoryId}:products`,
      async () => {
        const result = await this.db.query(
          'SELECT * FROM products WHERE category_id = $1',
          [categoryId]
        );
        return result.rows;
      },
      1800 // TTL: 30 minutes
    );
  }
}

/**
```

```
 * Write-through cache pattern
 */
class WriteThroughCache {
  constructor(
    private redis: Redis,
    private db: Pool
  ) {}

  async updateProduct(id: string, updates: any): Promise<void> {
    const client = await this.db.connect();

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

      // Step 1: Update database
      const result = await client.query(
        'UPDATE products SET name = $1, price = $2 WHERE id = $3 RETURNING *',
        [updates.name, updates.price, id]
      );

      const product = result.rows[0];

      await client.query('COMMIT');

      // Step 2: Update cache
      try {
        await this.redis.setex(
          `product:${id}`,
          3600,
          JSON.stringify(product)
        );
      } catch (cacheError) {
        // Cache update failed, invalidate instead
        console.error('[WriteThrough] Cache update failed, invalidating:',
cacheError);
        await this.redis.del(`product:${id}`).catch(() => {});
      }

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

/**
 * Event-driven invalidation
 */
import EventEmitter from 'events';
```

```
class EventDrivenCache extends EventEmitter {
  constructor(
    private redis: Redis,
    private db: Pool
  ) {
    super();
    this.setupListeners();
  }

  private setupListeners() {
    this.on('product:updated', async (productId: string) => {
      console.log(`[Event] Product ${productId} updated, invalidating caches`);

      // Invalidate related caches
      await this.redis.del(`product:${productId}`);

      // Invalidate category caches
      const result = await this.db.query(
        'SELECT category_id FROM products WHERE id = $1',
        [productId]
      );

      if (result.rows[0]) {
        const categoryId = result.rows[0].category_id;
        await this.redis.del(`category:${categoryId}:products`);
      }
    });
  }

  async updateProduct(id: string, updates: any): Promise<void> {
    await this.db.query(
      'UPDATE products SET name = $1, price = $2 WHERE id = $3',
      [updates.name, updates.price, id]
    );

    // Emit event for cache invalidation
    this.emit('product:updated', id);
  }
}

// Initialize
const redis = new Redis({ host: 'localhost', port: 6379 });
const db = new Pool({ host: 'localhost', database: 'myapp' });

const cacheManager = new CacheManager(redis, db);
const productRepo = new ProductRepository(db, cacheManager);

// Usage
app.get('/products/:id', async (req, res) => {
  const product = await productRepo.getProduct(req.params.id);
  res.json(product);
});
```

```
app.put('/products/:id', async (req, res) => {
  await productRepo.updateProduct(req.params.id, req.body);
  res.json({ success: true });
});

/**
 * Cache monitoring
 */
class CacheMonitor {
  private hits = 0;
  private misses = 0;

  recordHit() {
    this.hits++;
  }

  recordMiss() {
    this.misses++;
  }

  getHitRate(): number {
    const total = this.hits + this.misses;
    return total === 0 ? 0 : this.hits / total;
  }

  reset() {
    this.hits = 0;
    this.misses = 0;
  }
}

const monitor = new CacheMonitor();

app.get('/metrics/cache', (req, res) => {
  res.json({
    hitRate: monitor.getHitRate(),
    hits: monitor.hits,
    misses: monitor.misses,
  });
});
```

6. Correct Usage Patterns (Where This Shines)

Pattern 1: Cache-Aside (Default)

```
// Read: Check cache, fallback to DB
// Write: Update DB, delete cache
```

Pattern 2: Write-Through (Consistency Critical)

```
// Read: Check cache
// Write: Update DB + cache in transaction
```

Pattern 3: Event-Driven (Complex Dependencies)

```
// Write: Update DB, emit event
// Event handler: Invalidate all related caches
```

7. Failure Modes & Edge Cases

Cache Stampede

Problem: Cache expires, 1000 requests hit DB simultaneously.

Solution: Locking (only one request rebuilds cache).

Redis Down

Problem: Can't invalidate cache, app uses stale data.

Solution: Fallback to DB, short TTL.

Partial Invalidation

Problem: Some dependent caches not invalidated.

Solution: Dependency tracking, invalidate all related keys.

8. Performance Characteristics & Tradeoffs

Cache-Aside

- **Read:** Fast (cache hit), slow (cache miss)
- **Write:** Fast (just delete key)
- **Consistency:** Eventual

Write-Through

- **Read:** Fast (always cached)
- **Write:** Slow (2 operations)
- **Consistency:** Strong

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

Mistake 1: No Invalidation, Only TTL

Fix: Actively invalidate cache on writes.

Mistake 2: Global Cache Flush

Fix: Invalidate specific keys only.

Mistake 3: Forgotten Dependencies

Fix: Document and automate dependency tracking.

Mistake 4: Stale Data for Critical Operations

Fix: Never cache permissions, pricing with stale-while-revalidate.

Mistake 5: No Error Handling

Fix: Fallback to DB if cache fails.

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

Real-Time Data

Stock prices, live sports scores shouldn't be cached.

User-Specific Data

Don't cache data unique to each user (no benefit).

Frequently Updated Data

If data changes every second, caching adds complexity without benefit.

11. Related Concepts (With Contrast)

CDN Caching

Difference: CDN caches HTTP responses, application cache stores objects.

Database Query Cache

Difference: DB cache is automatic, application cache is manual.

Memoization

Related: In-memory function result caching (similar pattern).

12. Production Readiness Checklist

Cache Infrastructure

- [ ] Redis cluster (high availability)
- [ ] Cache keys namespaced
- [ ] TTL set on all keys
- [ ] Cache size monitoring

Invalidation

- [ ] Invalidate cache on writes
- [ ] Track dependencies
- [ ] Event-driven invalidation (if complex)

- [ ] Invalidation errors logged

Monitoring

- [ ] Cache hit/miss rate tracked
- [ ] Cache stampede detection
- [ ] Stale data alerts
- [ ] Cache memory usage

Safety

- [ ] Fallback to DB if cache fails
- [ ] No caching of critical data (permissions)
- [ ] No global cache flush in production
- [ ] Cache invalidation tested in staging