Thundering Herd & Cache Stampede

1. The Real Problem This Exists to Solve

When a heavily-accessed cached item expires, all concurrent requests simultaneously attempt to regenerate it, overwhelming the backend system. This creates a synchronized traffic spike that can take down databases or origin servers.

Real production scenario:

- Homepage data is cached in Redis with 5-minute TTL
- Cache serves 10,000 requests/second
- At t=300s, cache entry expires
- For the next 50-100ms (while cache is empty), ALL incoming requests miss cache
- 10,000 req/s × 0.1s = 1,000 requests simultaneously query the database
- Database query normally takes 20ms but with 1,000 concurrent queries, latency spikes to 5s
- Database connection pool (max 100) exhausted
- Queries queue up, timeout, clients retry
- 5,000 more requests arrive before first batch completes
- Database CPU at 100%, query latency at 30s+
- Database crashes or becomes unresponsive
- Cache remains empty (no successful query to repopulate it)
- All traffic now hits dead database
- **Complete outage**

The fundamental problem: **cache expiry creates synchronized miss patterns** that amplify load by orders of magnitude. What was 1 backend query per 5 minutes becomes 1,000+ simultaneous queries.

This differs from general load spikes because:

- It's predictable (happens at cache expiry)
- It's periodic (repeats every TTL interval)
- It's self-amplifying (slow queries → more retries → slower queries)
- It defeats caching entirely at the moment you need it most

2. The Naive / Incorrect Approaches (IMPORTANT)

❌ Incorrect Approach #1: Simple TTL-Based Caching

```
// Incorrect: Naive cache with fixed TTL
class CacheService {
  private redis: Redis;

  async get(key: string): Promise<Data | null> {
    const cached = await this.redis.get(key);
    if (cached) {
      return JSON.parse(cached);
    }

    // Cache miss: fetch from database
    const data = await database.query('SELECT * FROM data WHERE key = ?', [key]);
```

```
    // Store with 5 minute TTL
    await this.redis.setex(key, 300, JSON.stringify(data));

    return data;
  }
}
```

Why it seems reasonable:

- Standard caching pattern taught in tutorials
- Simple to implement
- Works well under low traffic
- Provides time-based cache invalidation

How it breaks:

- At second 300, cache expires completely (hard cutoff)
- Next 1,000 concurrent requests all see cache miss
- All 1,000 requests proceed to query database simultaneously
- Database cannot handle 1,000× normal load
- Queries slow down to 5-10s each
- Timeouts trigger, clients retry, amplifying the problem
- System oscillates between "working" and "overloaded" every 5 minutes

Production symptoms:

- Regular periodic latency spikes (every 5 minutes)
- Database CPU spikes at exact TTL boundaries
- Query queue depth spikes from 0 to 1000+ instantly
- Application logs show bursts of cache misses every N minutes
- Monitoring graphs show sawtooth pattern (flat then spike, repeating)
- Increasing traffic makes spikes worse (linear amplification)

❌ Incorrect Approach #2: Lazy Refresh Without Locking

```
// Incorrect: Attempt to refresh cache lazily, but no coordination
class CacheService {
  async get(key: string): Promise<Data> {
    const cached = await redis.get(key);
    if (cached) {
      return JSON.parse(cached);
    }

    // Cache miss: everyone fetches independently
    const data = await database.query('SELECT * FROM data WHERE key = ?', [key]);
    await redis.setex(key, 300, JSON.stringify(data));
    return data;
  }
}
```

Why it seems reasonable:

- Each request independently tries to repopulate cache

- Eventually one will succeed and others will benefit
- No explicit coordination needed

How it breaks:

- All concurrent requests see empty cache
- All proceed to database query simultaneously
- No deduplication of work
- Database receives N queries for the same data
- First query to complete writes to cache, but damage already done
- Subsequent queries' writes overwrite with identical data

Production symptoms:

- Database slow query log shows 100s of identical queries
- Redis shows multiple SETEX for same key within milliseconds
- Database connection pool exhaustion
- No improvement over Approach #1

❌ Incorrect Approach #3: Eager Refresh Without Stale-While-Revalidate

```
// Incorrect: Background refresh but cache empties during refresh
class CacheService {
  async get(key: string): Promise<Data> {
    const cached = await redis.get(key);

    if (cached) {
      const data = JSON.parse(cached);
      const ttl = await redis.ttl(key);

      // If expiring soon, refresh in background
      if (ttl < 30) {
        this.refreshInBackground(key);  // Fire and forget
      }

      return data;
    }

    // Cache miss: fetch from database
    const data = await database.query('SELECT * FROM data WHERE key = ?', [key]);
    await redis.setex(key, 300, JSON.stringify(data));
    return data;
  }

  private async refreshInBackground(key: string): Promise<void> {
    const data = await database.query('SELECT * FROM data WHERE key = ?', [key]);
    await redis.setex(key, 300, JSON.stringify(data));
  }
}
```

Why it seems reasonable:

- Proactively refreshes cache before expiry

- Avoids cold cache misses
- Background refresh doesn't block requests

How it breaks:

- Multiple concurrent requests within TTL < 30 window all trigger background refresh
- Still creates burst of duplicate queries
- If background refresh is slow (>30s), cache expires anyway
- Falls back to Approach #1 when background refresh can't keep up
- Race condition: cache expires while background refresh is in progress

Production symptoms:

- Background refresh reduces frequency of spikes but doesn't eliminate them
- Under high load, still see periodic database spikes
- Multiple background refreshes for same key (logged as duplicate queries)
- Cache sometimes empty despite background refresh running

❌ Incorrect Approach #4: Probabilistic Early Expiration Without Protection

```
// Incorrect: Probabilistic refresh but no stampede protection
class CacheService {
  async get(key: string): Promise<Data> {
    const cached = await redis.get(key);

    if (cached) {
      const data = JSON.parse(cached);
      const ttl = await redis.ttl(key);

      // Probabilistically refresh early (XFetch algorithm)
      const delta = 300 - ttl;  // Time since cache set
      const beta = 1.0;
      const shouldRefresh = Math.random() < (delta / (300 * beta));

      if (shouldRefresh) {
        // Refresh, but no coordination between requests
        const newData = await database.query('SELECT * FROM data WHERE key = ?',
[key]);
        await redis.setex(key, 300, JSON.stringify(newData));
        return newData;
      }

      return data;
    }

    // Cache miss
    const data = await database.query('SELECT * FROM data WHERE key = ?', [key]);
    await redis.setex(key, 300, JSON.stringify(data));
    return data;
  }
}
```

Why it seems reasonable:

- XFetch algorithm spreads out refresh times
- Reduces synchronized expiry
- Based on research paper

How it breaks:

- At high QPS, many requests within small time window all compute shouldRefresh
- Probability might be low (e.g., 1%), but at 10,000 req/s, that's 100 refreshes/second
- Still creates stampede, just smaller magnitude
- No deduplication of refresh work
- If refresh is expensive, many concurrent refreshes overload system

Production symptoms:

- Better than fixed TTL, but still periodic spikes
- Database load is more spread out but still elevated during refresh windows
- Multiple concurrent refresh queries in logs
- Spikes still visible at high traffic levels

3. Correct Mental Model (How It Actually Works)

Thundering herd is caused by two compounding factors:

1. **Synchronized expiry**: All clients discover cache miss at the same time
2. **Uncoordinated regeneration**: Each client independently tries to regenerate the cached value

The correct solution requires three key mechanisms working together:

Mechanism 1: Stale-While-Revalidate

Serve stale data while asynchronously refreshing in background:

```
Time 0: Cache entry created, TTL=300s
Time 290: Entry approaches expiry (soft TTL)
Time 291: First request:
  - Sees soft TTL expired
  - Returns stale data immediately (no blocking)
  - Triggers async refresh
Time 292-299: Subsequent requests:
  - See refresh in progress
  - Return stale data immediately
Time 300: Refresh completes, cache updated
Time 301: Hard TTL reached, old data expired
```

Key property: **Cache never empties**. There's always data to serve, even if stale.

Mechanism 2: Request Coalescing (Singleflight)

Multiple concurrent refresh attempts are deduplicated:

```
Request A: triggers refresh → starts database query
Request B: sees refresh in-flight → waits for A's result
Request C: sees refresh in-flight → waits for A's result
Request A completes → all three get the result
```

Only one database query is made regardless of concurrent request count.

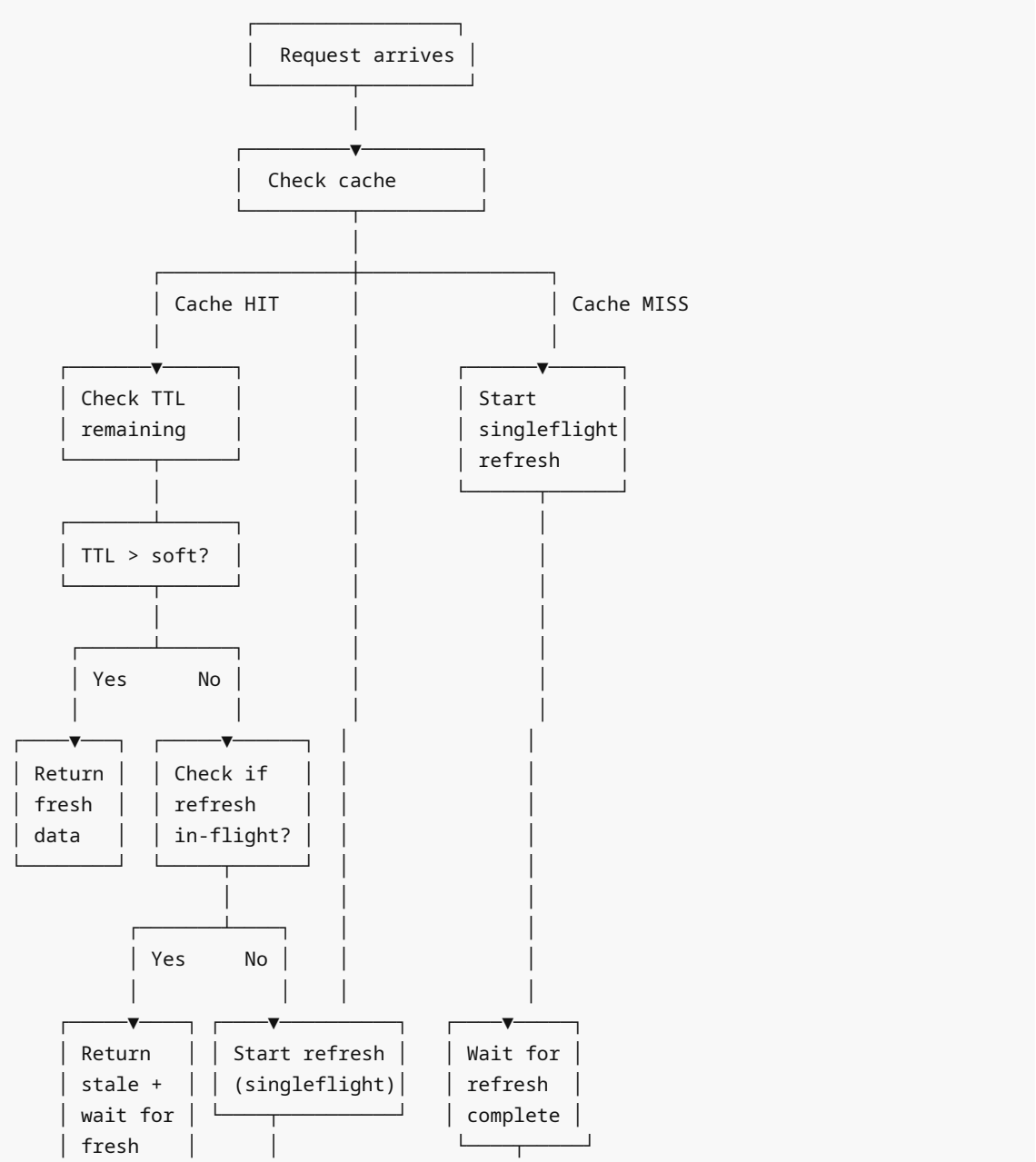
Mechanism 3: Probabilistic Early Expiration

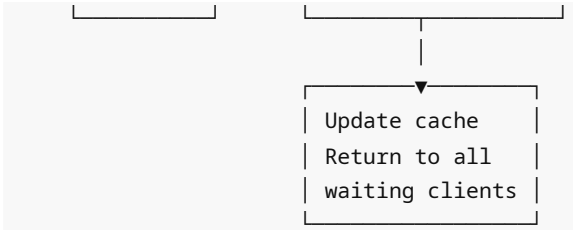
Spread out refresh times to avoid synchronized expiry across cache fleet:

```
Original TTL: 300s
Request 1 decides to refresh at: 285s
Request 2 decides to refresh at: 292s
Request 3 decides to refresh at: 278s

No synchronized expiry point.
```

Combined Flow





4. Correct Design & Algorithm

Dual-TTL Strategy

Use two TTL values:

- **Soft TTL**: When to start background refresh (e.g., 270s)
- **Hard TTL**: When to actually expire (e.g., 330s)

The 60-second window allows refresh to complete before data expires.

Algorithm

```
function get(key):
    cached = cache.get(key)

    if cached is null:
        return singleflight.execute(key, () => {
            data = fetchFromDatabase(key)
            cache.set(key, data, hardTTL=330, softTTL=270)
            return data
        })

    age = now() - cached.timestamp

    if age < cached.softTTL:
        return cached.data  // Fresh, return immediately

    if age < cached.hardTTL:
        // Stale but still usable
        asyncRefresh = singleflight.execute(key, () => {
            data = fetchFromDatabase(key)
            cache.set(key, data, hardTTL=330, softTTL=270)
            return data
        })

        // Return stale immediately, don't wait for refresh
        return cached.data

    // Expired, must wait for fresh data
    return singleflight.execute(key, () => {
        data = fetchFromDatabase(key)
        cache.set(key, data, hardTTL=330, softTTL=270)
```

```
        return data
    })
```

Probabilistic Refresh (Optional Enhancement)

Add randomness to soft TTL to spread refreshes:

```
softTTL = baseTTL * (0.9 + Math.random() * 0.2)  // ±10% jitter
```

This prevents synchronized refreshes across cache fleet even if all entries were populated at the same time.

5. Full Production-Grade Implementation

```
import { Redis } from 'ioredis';
import { Singleflight } from './singleflight';

interface CacheEntry<T> {
  data: T;
  timestamp: number;
  softTTL: number;
  hardTTL: number;
}

interface CacheOptions {
  softTTLSeconds: number;
  hardTTLSeconds: number;
  enableJitter: boolean;
  jitterPercent: number;
}

class ThunderingHerdProtectedCache {
  private redis: Redis;
  private singleflight: Singleflight;
  private readonly defaultOptions: CacheOptions = {
    softTTLSeconds: 270,
    hardTTLSeconds: 330,
    enableJitter: true,
    jitterPercent: 10,
  };

  constructor(redis: Redis) {
    this.redis = redis;
    this.singleflight = new Singleflight({
      maxSubscribers: 10000,
      maxDuration: 30000,
    });
  }

  /**
   * Get value from cache with thundering herd protection.
   *
```

```
   * - If data is fresh (age < soft TTL): return immediately
   * - If data is stale (soft TTL < age < hard TTL): return stale + async refresh
   * - If data is expired (age > hard TTL): wait for refresh
   * - If cache miss: fetch from origin with deduplication
   */
  async get<T>(
    key: string,
    fetchFn: () => Promise<T>,
    options?: Partial<CacheOptions>
  ): Promise<T> {
    const opts = { ...this.defaultOptions, ...options };

    // Try to get from cache
    const cached = await this.getCacheEntry<T>(key);

    if (!cached) {
      // Cache miss: fetch with singleflight protection
      return this.fetchAndCache(key, fetchFn, opts);
    }

    const age = Date.now() - cached.timestamp;

    // Case 1: Fresh data
    if (age < cached.softTTL * 1000) {
      return cached.data;
    }

    // Case 2: Stale but within hard TTL
    if (age < cached.hardTTL * 1000) {
      // Trigger async refresh (fire and forget)
      this.asyncRefresh(key, fetchFn, opts).catch(err => {
        console.error(`[Cache] Async refresh failed for ${key}:`, err);
      });

      // Return stale data immediately (no blocking)
      return cached.data;
    }

    // Case 3: Expired beyond hard TTL
    // Must wait for fresh data
    return this.fetchAndCache(key, fetchFn, opts);
  }

  /**
   * Fetch from origin and cache with singleflight deduplication
   */
  private async fetchAndCache<T>(
    key: string,
    fetchFn: () => Promise<T>,
    options: CacheOptions
  ): Promise<T> {
    return this.singleflight.execute(`cache:${key}`, async () => {
```

```
    const data = await fetchFn();
    await this.setCacheEntry(key, data, options);
    return data;
  });
}

/**
 * Async refresh: return stale immediately, refresh in background
 */
private async asyncRefresh<T>(
  key: string,
  fetchFn: () => Promise<T>,
  options: CacheOptions
): Promise<void> {
  // Use singleflight to deduplicate concurrent async refreshes
  await this.singleflight.execute(`refresh:${key}`, async () => {
    const data = await fetchFn();
    await this.setCacheEntry(key, data, options);
  });
}

/**
 * Get cache entry with metadata
 */
private async getCacheEntry<T>(key: string): Promise<CacheEntry<T> | null> {
  const raw = await this.redis.get(key);
  if (!raw) return null;

  try {
    return JSON.parse(raw);
  } catch (err) {
    console.error(`[Cache] Failed to parse cache entry for ${key}:`, err);
    return null;
  }
}

/**
 * Set cache entry with dual TTL and optional jitter
 */
private async setCacheEntry<T>(
  key: string,
  data: T,
  options: CacheOptions
): Promise<void> {
  const { softTTLSeconds, hardTTLSeconds, enableJitter, jitterPercent } = options;

  // Apply jitter to soft TTL to spread refresh times
  let finalSoftTTL = softTTLSeconds;
  if (enableJitter) {
    const jitterRange = softTTLSeconds * (jitterPercent / 100);
    const jitter = (Math.random() - 0.5) * 2 * jitterRange;
    finalSoftTTL = Math.floor(softTTLSeconds + jitter);
```

```
    }

    const entry: CacheEntry<T> = {
      data,
      timestamp: Date.now(),
      softTTL: finalSoftTTL,
      hardTTL: hardTTLSeconds,
    };

    // Store with hard TTL as Redis expiry
    await this.redis.setex(key, hardTTLSeconds, JSON.stringify(entry));
  }

  /**
   * Manually invalidate cache entry
   */
  async invalidate(key: string): Promise<void> {
    await this.redis.del(key);
  }

  /**
   * Warm cache proactively
   */
  async warm<T>(
    key: string,
    fetchFn: () => Promise<T>,
    options?: Partial<CacheOptions>
  ): Promise<void> {
    const opts = { ...this.defaultOptions, ...options };
    const data = await fetchFn();
    await this.setCacheEntry(key, data, opts);
  }

  /**
   * Get cache statistics for monitoring
   */
  getStats() {
    return this.singleflight.getStats();
  }
}

// Example usage
const redis = new Redis();
const cache = new ThunderingHerdProtectedCache(redis);

// Express API endpoint
app.get('/api/homepage', async (req, res) => {
  const data = await cache.get(
    'homepage-data',
    async () => {
      // Expensive database aggregation
      console.log('[DB] Fetching homepage data...');
```

```
      return await database.query(`
        SELECT * FROM posts
        ORDER BY created_at DESC
        LIMIT 100
      `);
    },
    {
      softTTLSeconds: 270,  // 4.5 minutes
      hardTTLSeconds: 330,  // 5.5 minutes
      enableJitter: true,
    }
  );

  res.json(data);
});

// Cache warming on deployment
async function warmCriticalCaches() {
  const criticalKeys = ['homepage-data', 'popular-posts', 'trending'];

  await Promise.all(
    criticalKeys.map(key =>
      cache.warm(key, () => fetchDataForKey(key))
    )
  );

  console.log('[Cache] Warmed critical caches');
}

// Monitoring endpoint
app.get('/metrics/cache', (req, res) => {
  const stats = cache.getStats();
  res.json({
    in_flight_refreshes: stats.inFlightCount,
    coalesced_requests: stats.totalSubscribers,
  });
});
```

6. Correct Usage Patterns (Where This Shines)

Pattern 1: High-Traffic Read-Heavy Endpoints

Homepage, popular content, trending items:

```
app.get('/api/trending', async (req, res) => {
  const trending = await cache.get(
    'trending-posts',
    async () => {
      return await db.query(`
        SELECT * FROM posts
        WHERE created_at > NOW() - INTERVAL 24 HOUR
```

```
        ORDER BY view_count DESC
        LIMIT 50
      `);
    },
    { softTTLSeconds: 240, hardTTLSeconds: 300 }
  );

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

Why this works:

- Trending page serves 100,000 requests/minute
- Database query takes 2 seconds (complex aggregation)
- Without protection: 100,000 × 2s = 200,000 seconds of DB time per cache miss
- With protection: 1 × 2s = 2 seconds of DB time, served to all waiters

Pattern 2: Expensive Aggregations

Analytics dashboards, reports:

```
async function getDashboardData(userId: string): Promise<Dashboard> {
  return cache.get(
    `dashboard:${userId}`,
    async () => {
      // 10-second aggregation across multiple tables
      const [revenue, orders, customers] = await Promise.all([
        db.query('SELECT SUM(amount) FROM orders WHERE user_id = ?', [userId]),
        db.query('SELECT COUNT(*) FROM orders WHERE user_id = ?', [userId]),
        db.query('SELECT COUNT(*) FROM customers WHERE seller_id = ?', [userId]),
      ]);

      return { revenue, orders, customers };
    },
    { softTTLSeconds: 1800, hardTTLSeconds: 2100 }  // 30min soft, 35min hard
  );
}
```

Why this works:

- Dashboard refresh is expensive (10 seconds)
- Users tolerate slightly stale data (30 minutes old)
- Stale-while-revalidate ensures users never wait 10 seconds
- Singleflight prevents multiple concurrent 10s queries

Pattern 3: External API Calls with Rate Limits

Third-party API integration:

```
async function getGitHubStars(repo: string): Promise<number> {
  return cache.get(
    `github:stars:${repo}`,
    async () => {
```

```
      const response = await fetch(
        `https://api.github.com/repos/${repo}`,
        { headers: { Authorization: `token ${GITHUB_TOKEN}` } }
      );
      const data = await response.json();
      return data.stargazers_count;
    },
    { softTTLSeconds: 3600, hardTTLSeconds: 4200 }  // 1 hour
  );
}
```

Why this works:

- GitHub API has rate limits (5000 req/hour)
- Without protection: 10,000 users viewing same repo = 10,000 API calls
- With protection: 1 API call per hour regardless of traffic
- Stays within rate limits even during traffic spikes

Pattern 4: Session/Auth Data

User profiles, permissions:

```
async function getUserPermissions(userId: string): Promise<Permissions> {
  return cache.get(
    `permissions:${userId}`,
    async () => {
      return await db.query(`
        SELECT p.* FROM permissions p
        JOIN user_roles ur ON ur.role_id = p.role_id
        WHERE ur.user_id = ?
      `, [userId]);
    },
    { softTTLSeconds: 270, hardTTLSeconds: 330 }
  );
}
```

Why this works:

- Every request needs auth check
- Permission changes are rare (minutes to hours)
- Stale data is acceptable (eventual consistency)
- Prevents auth DB from being bottleneck

7. Failure Modes & Edge Cases

Cold Start Stampede

Problem: Application restart with empty cache, all keys miss simultaneously.

Symptoms:

- Application starts, all caches empty
- First wave of requests all miss cache

- Database receives 100× normal load
- Application cannot serve traffic for minutes

Mitigation:

- Cache warming on startup
- Gradual traffic ramp-up
- Circuit breaker to shed load during cold start

```
async function startup() {
  console.log('[Startup] Warming cache...');
  await warmCriticalCaches();
  console.log('[Startup] Cache warm, ready for traffic');
}
```

Refresh Timeout Causing Hard Expiry

Problem: Background refresh takes longer than (hard TTL - soft TTL) window, cache expires during refresh.

Symptoms:

- Async refresh starts at soft TTL
- Refresh takes 120 seconds
- Hard TTL reached before refresh completes
- Next requests see cache miss, trigger new stampede

Mitigation:

- Set hard TTL with enough buffer (soft + 2× max refresh time)
- Monitor refresh duration, alert if approaching window
- If refresh is slow, serve stale indefinitely until refresh completes

```
// Extended hard TTL for slow refreshes
{
  softTTLSeconds: 270,      // 4.5 min
  hardTTLSeconds: 600,      // 10 min (5.5 min buffer)
}
```

Singleflight Stuck Entry

Problem: Refresh starts, query hangs, all future requests wait forever on stuck promise.

Symptoms:

- Singleflight entry remains in-flight for minutes
- All requests for that key timeout
- Manually restarting application fixes it

Mitigation:

- Set max duration for singleflight entries (e.g., 30s)
- If entry exceeds max duration, reject and allow new request to retry
- Implement query timeouts in fetch function

```
this.singleflight = new Singleflight({
  maxDuration: 30000,  // 30 seconds max
});
```

Redis Unavailability

Problem: Redis goes down, all cache reads fail, full traffic hits database.

Symptoms:

- Redis connection errors
- Every request becomes cache miss
- Database overload

Mitigation:

- In-memory fallback cache (LRU)
- Circuit breaker to serve stale from local memory
- Graceful degradation to read-only mode

```
async get<T>(key: string, fetchFn: () => Promise<T>): Promise<T> {
  try {
    return await this.getFromRedis(key, fetchFn);
  } catch (redisError) {
    console.error('[Cache] Redis error, using local fallback:', redisError);
    return await this.getFromLocalCache(key, fetchFn);
  }
}
```

Cache Poisoning from Bad Refresh

Problem: Background refresh fetches bad data, caches it, serves to all users.

Symptoms:

- One bad database query result (null, error data)
- Gets cached with full TTL
- All users see bad data for next 5 minutes

Mitigation:

- Validate data before caching
- Don't cache null or error results
- Implement health checks for cached data

```
const data = await fetchFn();

// Validate before caching
if (!data || !isValid(data)) {
  console.warn('[Cache] Refusing to cache invalid data for', key);
  throw new Error('Invalid data');
}
```

```
await this.setCacheEntry(key, data, options);
```

8. Performance Characteristics & Tradeoffs

Latency Impact

Without protection:

- p50: 10ms (cache hit)
- p99: 2000ms (cache miss, database query)
- During stampede: p99 > 30,000ms (database overload)

With protection:

- p50: 10ms (cache hit, fresh data)
- p90: 10ms (cache hit, stale data served immediately)
- p99: 15ms (cache hit, stale + async refresh)
- p99.9: 2000ms (hard expiry, wait for refresh)

Improvement: 15× better p99 during cache expiry windows

Throughput Impact

Without protection:

- Maximum sustainable QPS = database max QPS / cache miss rate
- During stampede: effective QPS drops to ~0 (timeouts)

With protection:

- Maximum QPS = cache layer throughput (Redis: 100k+ QPS)
- During refresh: no throughput impact (async)

Improvement: 100× higher sustainable throughput

Memory Overhead

Cache entries:

- Each entry stores: data + timestamp + soft TTL + hard TTL
- Overhead: ~20 bytes per entry
- For 1M entries: 20MB overhead (negligible)

Singleflight:

- Stores in-flight refresh operations
- Typically < 100 concurrent refreshes
- Overhead: < 10KB

CPU Overhead

Per request:

- Timestamp comparison: <1µs
- JSON parse: 1-10µs
- Redis GET: 0.5ms

Total overhead: < 1ms per request (negligible compared to application logic)

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

Mistake 1: Soft TTL > Hard TTL

Why engineers do it: Typo or misunderstanding.

What breaks: Cache always serves stale, never refreshes, or hard expiry happens before soft expiry.

```
// WRONG
{
  softTTLSeconds: 600,   // 10 minutes
  hardTTLSeconds: 300,   // 5 minutes
}
// Hard TTL expires first, defeating stale-while-revalidate
```

Detection: Monitoring shows cache entries expiring before refresh triggers.

Fix: Ensure soft TTL < hard TTL with sufficient buffer.

```
// Correct
{
  softTTLSeconds: 270,   // 4.5 minutes
  hardTTLSeconds: 330,   // 5.5 minutes (1 minute buffer)
}
```

Mistake 2: Not Using Singleflight for Refresh

Why engineers do it: Implement stale-while-revalidate but forget deduplication.

What breaks: 1000 concurrent requests trigger 1000 async refreshes.

```
// WRONG
if (isStale) {
  // Each request triggers independent refresh
  this.refresh(key, fetchFn);  // 1000 refreshes!
  return staleData;
}
```

Detection: Database logs show multiple identical queries during refresh window.

Fix: Wrap refresh in singleflight.

```
// Correct
if (isStale) {
  this.singleflight.execute(`refresh:${key}`, () => this.refresh(key, fetchFn));
  return staleData;
}
```

Mistake 3: Serving Stale Indefinitely

Why engineers do it: "Stale data is better than no data."

What breaks: Database is down for hours, application serves increasingly stale data, business logic breaks.

```
// WRONG: No hard TTL enforcement
if (cached) {
  // Always return cached data regardless of age
  this.asyncRefresh(key, fetchFn);
  return cached.data;
}
```

Detection: Monitoring shows some cache entries are days old.

Fix: Enforce hard TTL, fail requests beyond it.

```
// Correct
if (age > hardTTL) {
  // Too stale, must wait for fresh data
  return await this.fetchAndCache(key, fetchFn);
}
```

Mistake 4: Not Handling Refresh Failures

Why engineers do it: Assume refresh always succeeds.

What breaks: Refresh fails, cache expires, future requests stampede.

```
// WRONG
async asyncRefresh(key, fetchFn) {
  const data = await fetchFn();  // Throws on error
  await this.setCacheEntry(key, data);
}
// Error leaves cache expired
```

Detection: Logs show refresh errors, followed by stampede.

Fix: On refresh failure, extend TTL to retry later.

```
// Correct
async asyncRefresh(key, fetchFn) {
  try {
    const data = await fetchFn();
    await this.setCacheEntry(key, data, options);
  } catch (error) {
    console.error('[Cache] Refresh failed, extending TTL:', error);
    // Extend TTL to retry in 30 seconds
    await this.redis.expire(key, 30);
  }
}
```

Mistake 5: Cache Key Collision

Why engineers do it: Use simple keys without namespacing.

What breaks: Different data types share same key, wrong data returned.

```
// WRONG
cache.get('user:123', () => getUserProfile(123));
cache.get('user:123', () => getUserOrders(123));
// Second call returns profile data!
```

Detection: Users report seeing wrong data inconsistently.

Fix: Include data type in key.

```
// Correct
cache.get('user:profile:123', () => getUserProfile(123));
cache.get('user:orders:123', () => getUserOrders(123));
```

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

Anti-Pattern 1: Real-Time Data

Don't cache data that must be up-to-the-second accurate:

```
// WRONG: Stock prices must be real-time
cache.get('stock:AAPL', () => getStockPrice('AAPL'));
// User sees stale price from 5 minutes ago
```

Anti-Pattern 2: Write-Heavy Data

Don't cache data that changes frequently:

```
// WRONG: User's shopping cart changes on every action
cache.get(`cart:${userId}`, () => getCart(userId));
// Add item → cache invalidated → stampede on next 1000 page views
```

Anti-Pattern 3: User-Specific Private Data

Don't cache when data varies per user:

```
// WRONG: Each user has unique data
cache.get('notifications', () => getNotifications(userId));
// All users share same cache key → privacy violation
```

Include userId in key, but be aware this reduces cache hit rate.

Anti-Pattern 4: Small, Fast Queries

Don't cache if database query is faster than cache lookup:

```
// WRONG: Primary key lookup is 0.1ms
cache.get(`user:${id}`, () => db.getUserById(id));
// Cache lookup: 0.5ms (Redis network) + 0.1ms (parsing) = 0.6ms
// Direct DB: 0.1ms (local connection pool)
// Cache is slower!
```

11. Related Concepts (With Contrast)

Request Coalescing (Singleflight)

Difference: Singleflight deduplicates concurrent identical requests. Thundering herd protection prevents cache stampedes using stale-while-revalidate + singleflight.

Relationship: Thundering herd protection uses singleflight as a component.

Cache Warming

Difference: Cache warming proactively populates cache before traffic arrives. Thundering herd protection handles cache expiry during traffic.

When to combine: Warm cache on startup (prevent cold start stampede), use thundering herd protection for ongoing operation.

Circuit Breaker

Difference: Circuit breaker stops calling failing services. Thundering herd protection prevents overloading working services during cache miss.

When to combine: Use circuit breaker to detect backend overload, use thundering herd protection to prevent causing overload.

CDN / Edge Caching

Difference: CDN caches at network edge (geographic distribution). Thundering herd protection applies at application layer.

When to combine: Use both. CDN prevents stampedes at edge, application-layer cache handles origin protection.

12. Production Readiness Checklist

Metrics to Monitor

- [ ] Cache hit rate (should be >95%)
- [ ] Cache miss rate during stampede windows
- [ ] Async refresh success rate
- [ ] Async refresh latency (p50, p99)
- [ ] Number of in-flight singleflight operations
- [ ] Number of coalesced requests (singleflight subscribers)
- [ ] Hard TTL expiry rate (should be near 0%)
- [ ] Database query count before/after (should drop 100×)

Logging Requirements

- [ ] Log cache misses with key and reason
- [ ] Log async refresh start/complete with duration
- [ ] Log async refresh failures with error details
- [ ] Log when hard TTL is reached (should be rare)
- [ ] Log singleflight deduplication stats

Configuration

- [ ] Set soft TTL to ~90% of hard TTL
- [ ] Set hard TTL with 2× refresh duration buffer
- [ ] Enable jitter (±10-20%)
- [ ] Set singleflight max duration to 2× expected refresh time
- [ ] Set singleflight max subscribers based on peak QPS

Load Testing

- [ ] Test cache miss stampede (1000 concurrent requests, empty cache)
- [ ] Verify only 1 database query is made
- [ ] Test cache expiry stampede (TTL expires, 1000 concurrent requests)
- [ ] Verify stale data is served, async refresh triggered once
- [ ] Test refresh failure (database returns error)
- [ ] Verify cache TTL is extended, retry happens
- [ ] Measure latency during refresh (should not increase)

Rollout Strategy

- [ ] Deploy to staging, monitor for 1 hour
- [ ] Deploy to 1% of production traffic
- [ ] Monitor database load reduction
- [ ] Gradually increase to 10%, 50%, 100%
- [ ] Have kill switch to disable caching instantly

Alerting

- [ ] Alert if cache hit rate < 90%
- [ ] Alert if async refresh failure rate > 1%
- [ ] Alert if refresh latency > (hard TTL - soft TTL)
- [ ] Alert if hard TTL expiry rate > 1%
- [ ] Alert if singleflight max duration exceeded
- [ ] Alert if database query count increases unexpectedly