Chapter 5: Caching

1. What is Caching and When to Use It

Definition

Caching is the technique of storing copies of data in a temporary, fast-access storage layer so that future requests for that data can be served faster.

Analogy: Think of your desk while studying.

Without Cache (Going to Library Every Time):

```
┌──────────────────────────────────────────┐
│ Need info → Walk to library (5 min)      │
│ Find book → Read → Walk back (5 min)     │
│ Total time: 10 minutes per lookup        │
└──────────────────────────────────────────┘
```

With Cache (Keeping Books on Desk):

```
┌──────────────────────────────────────────┐
│ Need info → Check desk (5 seconds)       │
│ If book there → Read immediately         │
│ Total time: 5 seconds per lookup         │
│                                          │
│ 120x faster! (600s vs 5s)                │
└──────────────────────────────────────────┘
```

The Cache Hierarchy

Speed & Cost Hierarchy (Top = Fastest & Most Expensive):

```
┌──────────────────────────────────────────────────────────────┐
│ CPU L1 Cache      │ 0.5 ns  │ 32 KB   │ $$$$$  │
│ CPU L2 Cache      │ 7 ns    │ 256 KB  │ $$$$   │
│ CPU L3 Cache      │ 15 ns   │ 8 MB    │ $$$    │
├──────────────────────────────────────────────────────────────┤
│ RAM (Memory)      │ 100 ns  │ 16 GB   │ $$     │ ← Application Cache
├──────────────────────────────────────────────────────────────┤
│ SSD               │ 150 µs  │ 1 TB    │ $      │
│ Hard Disk         │ 10 ms   │ 10 TB   │ ¢      │ ← Database
└──────────────────────────────────────────────────────────────┘
```

Application-level caching (our focus) typically uses RAM

RAM is ~1,000x faster than SSD

RAM is ~100,000x faster than Hard Disk

Real-World Performance Impact

Example: E-commerce Product Page

Without Caching:

User requests product page

↓

Query database for product details (50ms)

↓

Query database for reviews (30ms)

↓

Query database for recommendations (40ms)

↓

Query database for inventory (20ms)

↓

Total: 140ms response time

With Caching:

User requests product page

↓

Check cache (0.5ms)

↓

Cache hit! Return data

↓

Total: 0.5ms response time

Result: 280x faster! (140ms → 0.5ms)

Code Example:

python

```python
import time
import random

# Simulate database query (slow)
def get_product_from_database(product_id):
    """Simulate database query - takes 50ms"""
    time.sleep(0.05)  # 50ms delay
    return {
        'id': product_id,
        'name': f'Product {product_id}',
        'price': random.randint(10, 100)
    }

# Without cache
def get_product_no_cache(product_id):
    start = time.time()
    product = get_product_from_database(product_id)
    elapsed = (time.time() - start) * 1000
    print(f"No cache: {elapsed:.2f}ms")
    return product

# With cache
product_cache = {}

def get_product_with_cache(product_id):
    start = time.time()

    # Check cache first
    if product_id in product_cache:
        elapsed = (time.time() - start) * 1000
        print(f"Cache hit: {elapsed:.2f}ms")
        return product_cache[product_id]

    # Cache miss - fetch from database
    product = get_product_from_database(product_id)
    product_cache[product_id] = product

    elapsed = (time.time() - start) * 1000
    print(f"Cache miss: {elapsed:.2f}ms")
    return product

# Test: Request same product 5 times
print("Without Cache:")
for i in range(5):
    get_product_no_cache(123)
```

```
print("\nWith Cache:")
for i in range(5):
    get_product_with_cache(123)

# Output:
# Without Cache:
# No cache: 50.12ms
# No cache: 50.08ms
# No cache: 50.11ms
# No cache: 50.09ms
# No cache: 50.10ms
#
# With Cache:
# Cache miss: 50.15ms (first time - has to fetch)
# Cache hit: 0.002ms   (subsequent - from cache)
# Cache hit: 0.001ms
# Cache hit: 0.001ms
# Cache hit: 0.001ms
```

When to Use Caching

✓ GOOD CANDIDATES FOR CACHING:

1. Frequently accessed data
   Example: Popular products, trending posts

2. Expensive computations
   Example: Recommendation algorithms, reports

3. Slow database queries
   Example: Complex joins, aggregations

4. Static or rarely changing data
   Example: Configuration, country lists

5. High read-to-write ratio
   Example: 90% reads, 10% writes

✗ BAD CANDIDATES FOR CACHING:

1. Frequently changing data
   Example: Stock prices, live sports scores

2. Data that must be fresh
   Example: Bank account balance

3. Large datasets with low access frequency
   Example: Old archived data

4. Highly personalized data
   Example: User-specific calculations (unless cached per user)

5. Security-sensitive data
   Example: Passwords, credit cards

Real-World Scenarios:

Scenario 1: News Website
———————————————————

Homepage: 1 million views/hour
Updates: New articles every 15 minutes

Solution: Cache homepage for 15 minutes
Result:
- Without cache: 1M database queries/hour
- With cache: 4 database queries/hour (1 per 15 min)
- 250,000x reduction in database load!

Scenario 2: Social Media Feed
———————————————————

User's feed: Combination of posts from 500 friends
Query time: 2 seconds (complex aggregation)
User checks feed: 20 times/day

Solution: Cache user's feed for 5 minutes
Result:
- Without cache: 20 × 2s = 40 seconds of database work per user
- With cache: ~4 × 2s = 8 seconds (refresh every 5 min)
- 5x reduction in database load per user

Scenario 3: API Rate Limiting
———————————————————

Need to track requests per user
Check limit: On every API call

Solution: Cache request count in Redis
Result:
- Without cache: Database query on every request
- With cache: In-memory counter, no database queries
- Infinite improvement (0 database queries)

2. Cache Hit vs Cache Miss

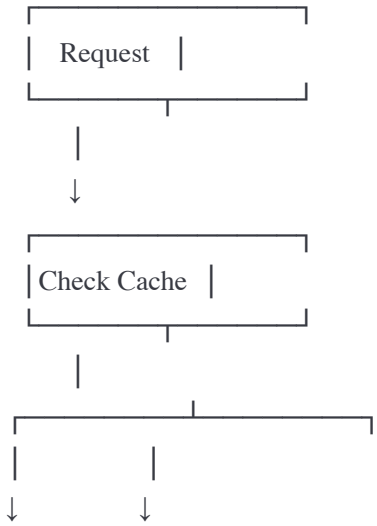
Understanding these metrics is crucial for cache performance.

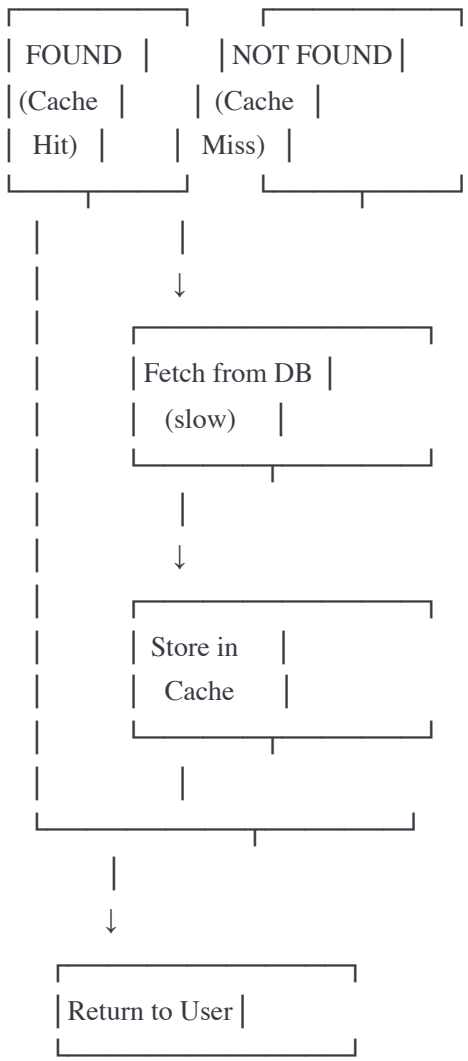
Definitions

```
┌──────────────────────────────────────────────────────────────┐
│                        CACHE HIT                             │
├──────────────────────────────────────────────────────────────┤
│ Request for data that IS in cache                            │
│                                                              │
│ User requests → Cache → Found! → Return                      │
│                                                              │
│ Fast! (< 1ms typically)                                      │
└──────────────────────────────────────────────────────────────┘


┌──────────────────────────────────────────────────────────────┐
│                        CACHE MISS                            │
├──────────────────────────────────────────────────────────────┤
│ Request for data that is NOT in cache                        │
│                                                              │
│ User requests → Cache → Not found                            │
│                 → Fetch from database                        │
│                 → Store in cache                             │
│                 → Return                                     │
│                                                              │
│ Slow! (same as no cache + cache write overhead)              │
└──────────────────────────────────────────────────────────────┘
```

Visual Flow

Request Flow with Caching:





Cache Hit Ratio

Formula:

Cache Hit Ratio = Cache Hits / Total Requests × 100%

Example:
Total requests: 10,000
Cache hits: 9,000
Cache misses: 1,000

Hit ratio = 9,000 / 10,000 = 90%
Miss ratio = 1,000 / 10,000 = 10%

Impact of Hit Ratio:

Scenario: 10,000 requests per second
Database query time: 50ms
Cache query time: 1ms

With 90% hit ratio:
- Cache hits: 9,000 × 1ms = 9 seconds
- Cache misses: 1,000 × 50ms = 50 seconds
- Total: 59 seconds of processing time

With 99% hit ratio:
- Cache hits: 9,900 × 1ms = 9.9 seconds
- Cache misses: 100 × 50ms = 5 seconds
- Total: 14.9 seconds of processing time

Just improving from 90% to 99% hit ratio:
→ 4x improvement! (59s → 14.9s)

Code to Track Cache Metrics:

python

```
from collections import defaultdict
import time

class CacheWithMetrics:
    def __init__(self):
        self.cache = {}
        self.hits = 0
        self.misses = 0
        self.total_hit_time = 0
        self.total_miss_time = 0

    def get(self, key, fetch_function):
        """
        Get value from cache or fetch it
        fetch_function: function to call on cache miss
        """
        start = time.time()

        if key in self.cache:
            # Cache hit
            value = self.cache[key]
            elapsed = time.time() - start

            self.hits += 1
            self.total_hit_time += elapsed

            return value
        else:
            # Cache miss
            value = fetch_function(key)
            self.cache[key] = value
            elapsed = time.time() - start

            self.misses += 1
            self.total_miss_time += elapsed

            return value

    def get_stats(self):
        """Return cache performance statistics"""
        total = self.hits + self.misses
        if total == 0:
            return "No requests yet"

        hit_ratio = (self.hits / total) * 100
        miss_ratio = (self.misses / total) * 100
```

```
        avg_hit_time = (self.total_hit_time / self.hits * 1000) if self.hits > 0 else 0
        avg_miss_time = (self.total_miss_time / self.misses * 1000) if self.misses > 0 else 0

        return f"""
Cache Statistics:
─────────────────────────
Total Requests: {total:,}
Cache Hits: {self.hits:,} ({hit_ratio:.2f}%)
Cache Misses: {self.misses:,} ({miss_ratio:.2f}%)

Performance:
Average hit time: {avg_hit_time:.3f}ms
Average miss time: {avg_miss_time:.3f}ms
Speedup: {avg_miss_time/avg_hit_time:.1f}x faster on hits
        """

# Example usage
cache = CacheWithMetrics()

def slow_database_query(user_id):
    """Simulate slow database"""
    time.sleep(0.05)  # 50ms
    return f"User data for {user_id}"

# Simulate traffic
import random
user_ids = [1, 2, 3, 4, 5]  # 5 users

# 1000 requests with realistic distribution (some users more popular)
for _ in range(1000):
    # 80% requests go to users 1-3 (popular users)
    # 20% requests go to users 4-5 (less popular)
    if random.random() < 0.8:
        user_id = random.choice([1, 2, 3])
    else:
        user_id = random.choice([4, 5])

    cache.get(user_id, slow_database_query)

print(cache.get_stats())

# Output:
# Cache Statistics:
# ─────────────────────────
# Total Requests: 1,000
# Cache Hits: 995 (99.50%)
```

```
# Cache Misses: 5 (0.50%)
#
# Performance:
# Average hit time: 0.001ms
# Average miss time: 50.234ms
# Speedup: 50234.0x faster on hits
```

Improving Cache Hit Ratio

```
Strategies to Improve Hit Ratio:

1. INCREASE CACHE SIZE
   More items can fit → fewer evictions → more hits
   Trade-off: More memory usage

2. BETTER EVICTION POLICY
   Keep frequently accessed items → more hits
   (We'll cover LRU, LFU later)

3. LONGER TTL (Time To Live)
   Items stay in cache longer → more hits
   Trade-off: Stale data

4. PREWARMING/PRELOADING
   Load popular items before requests
   Example: Load top 100 products at startup

5. CACHE HIERARCHY
   Multiple cache levels (L1, L2)
   Fast small cache + slower large cache

6. INTELLIGENT PREFETCHING
   Predict what user will request next
   Example: Preload next page of results
```

---

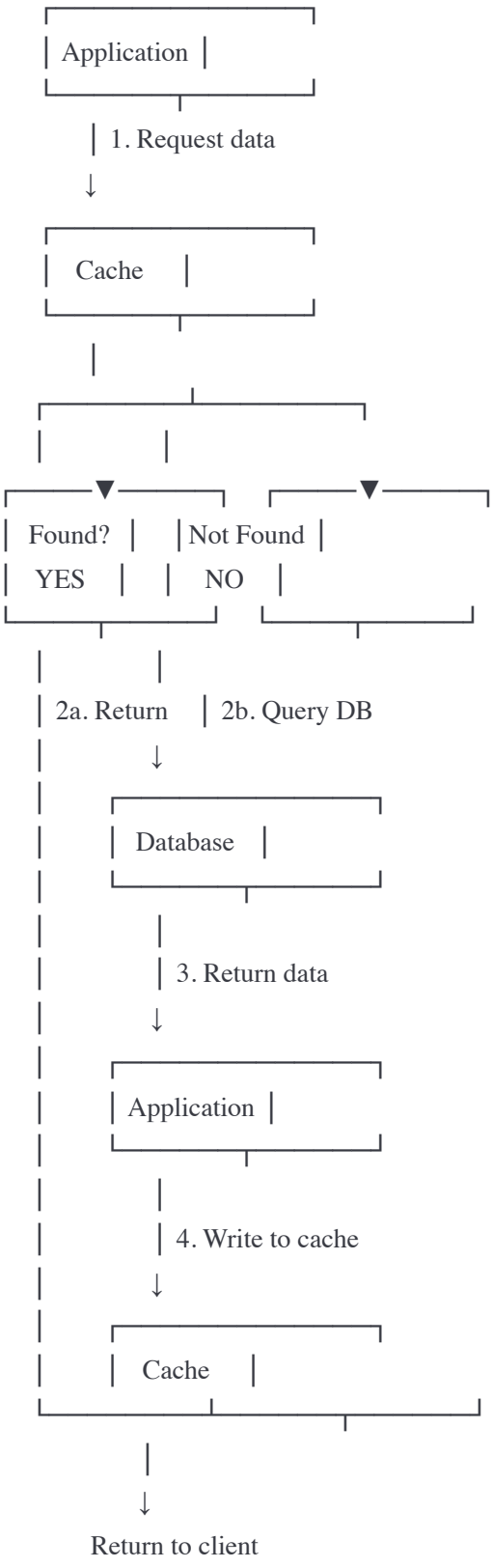
3. Caching Strategies

Different strategies for different use cases.

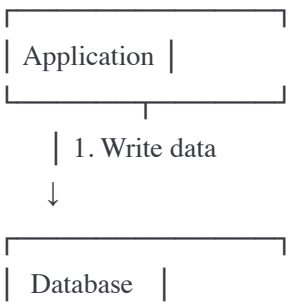
1. Cache-Aside (Lazy Loading)

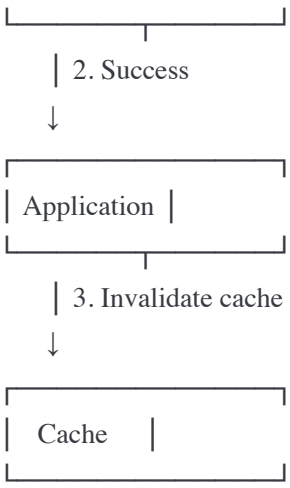
Most common pattern. Application manages the cache.

Read Flow:



Write Flow:





Python Implementation:

python

```python
import redis
import psycopg2
import json

class CacheAsidePattern:
    def __init__(self):
        self.cache = redis.Redis(host='localhost', port=6379, decode_responses=True)
        self.db = psycopg2.connect(
            host="localhost",
            database="myapp",
            user="user",
            password="password"
        )

    def get_user(self, user_id):
        """Cache-aside read pattern"""
        cache_key = f"user:{user_id}"

        # 1. Try to get from cache
        cached_user = self.cache.get(cache_key)

        if cached_user:
            print(f"Cache HIT for user {user_id}")
            return json.loads(cached_user)

        print(f"Cache MISS for user {user_id}")

        # 2. Cache miss - fetch from database
        cursor = self.db.cursor()
        cursor.execute(
            "SELECT id, name, email FROM users WHERE id = %s",
            (user_id,)
        )
        row = cursor.fetchone()

        if not row:
            return None

        user = {
            'id': row[0],
            'name': row[1],
            'email': row[2]
        }

        # 3. Store in cache (TTL: 1 hour)
        self.cache.setex(
```

```
            cache_key,
            3600,  # 1 hour
            json.dumps(user)
        )

        return user

    def update_user(self, user_id, data):
        """Cache-aside write pattern"""
        cache_key = f"user:{user_id}"

        # 1. Update database
        cursor = self.db.cursor()
        cursor.execute(
            "UPDATE users SET name = %s, email = %s WHERE id = %s",
            (data['name'], data['email'], user_id)
        )
        self.db.commit()

        # 2. Invalidate cache
        self.cache.delete(cache_key)

        print(f"Updated user {user_id} and invalidated cache")

        # Note: Next read will cache miss and reload from DB

# Usage
cache_aside = CacheAsidePattern()

# First read - cache miss
user = cache_aside.get_user(123)  # Cache MISS - fetches from DB

# Second read - cache hit
user = cache_aside.get_user(123)  # Cache HIT - instant!

# Update user
cache_aside.update_user(123, {'name': 'John Updated', 'email': 'john@example.com'})

# Next read - cache miss (was invalidated)
user = cache_aside.get_user(123)  # Cache MISS - fetches updated data
```

Pros and Cons:

✓ ADVANTAGES:

- Cache contains only requested data (efficient)
- Cache failure doesn't break system (degraded performance only)
- Easy to implement
- Works well for read-heavy workloads

✗ DISADVANTAGES:

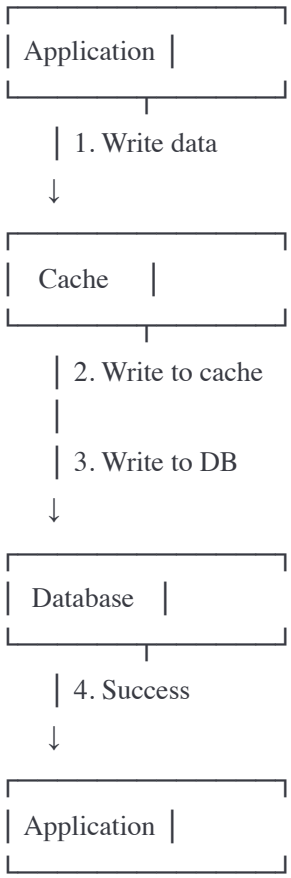
- Cache miss penalty (3 round trips: check cache, read DB, write cache)
- Potential for stale data
- Initial requests always slow (empty cache)
- Cache and DB can be inconsistent

---

2. Write-Through Cache

Cache and database updated together. Always consistent.

Write Flow:



Every write goes through cache to database

Cache always has latest data

Python Implementation:

python

```python
class WriteThroughCache:
    def __init__(self):
        self.cache = redis.Redis(host='localhost', port=6379, decode_responses=True)
        self.db = psycopg2.connect(...)

    def get_user(self, user_id):
        """Read from cache (always present if exists)"""
        cache_key = f"user:{user_id}"

        # Check cache first
        cached_user = self.cache.get(cache_key)
        if cached_user:
            return json.loads(cached_user)

        # Not in cache - check database
        cursor = self.db.cursor()
        cursor.execute(
            "SELECT id, name, email FROM users WHERE id = %s",
            (user_id,)
        )
        row = cursor.fetchone()

        if row:
            user = {'id': row[0], 'name': row[1], 'email': row[2]}
            # Populate cache
            self.cache.setex(cache_key, 3600, json.dumps(user))
            return user

        return None

    def update_user(self, user_id, data):
        """Write-through: Update cache AND database together"""
        cache_key = f"user:{user_id}"

        try:
            # 1. Update cache first
            self.cache.setex(
                cache_key,
                3600,
                json.dumps({
                    'id': user_id,
                    'name': data['name'],
                    'email': data['email']
                })
            )
```

```
        # 2. Update database
        cursor = self.db.cursor()
        cursor.execute(
            "UPDATE users SET name = %s, email = %s WHERE id = %s",
            (data['name'], data['email'], user_id)
        )
        self.db.commit()

        print(f"Write-through complete for user {user_id}")

    except Exception as e:
        # Rollback on error
        self.cache.delete(cache_key)
        self.db.rollback()
        raise e

def create_user(self, user_id, data):
    """Write-through: Create in cache AND database"""
    cache_key = f"user:{user_id}"

    try:
        # 1. Write to database first (source of truth)
        cursor = self.db.cursor()
        cursor.execute(
            "INSERT INTO users (id, name, email) VALUES (%s, %s, %s)",
            (user_id, data['name'], data['email'])
        )
        self.db.commit()

        # 2. Immediately add to cache
        self.cache.setex(
            cache_key,
            3600,
            json.dumps({
                'id': user_id,
                'name': data['name'],
                'email': data['email']
            })
        )

        print(f"Created user {user_id} in DB and cache")

    except Exception as e:
        self.db.rollback()
        self.cache.delete(cache_key)
        raise e
```

Pros and Cons:

✓ ADVANTAGES:
- Cache always consistent with database
- Read performance excellent (always in cache)
- No stale data issues
- Simpler to reason about

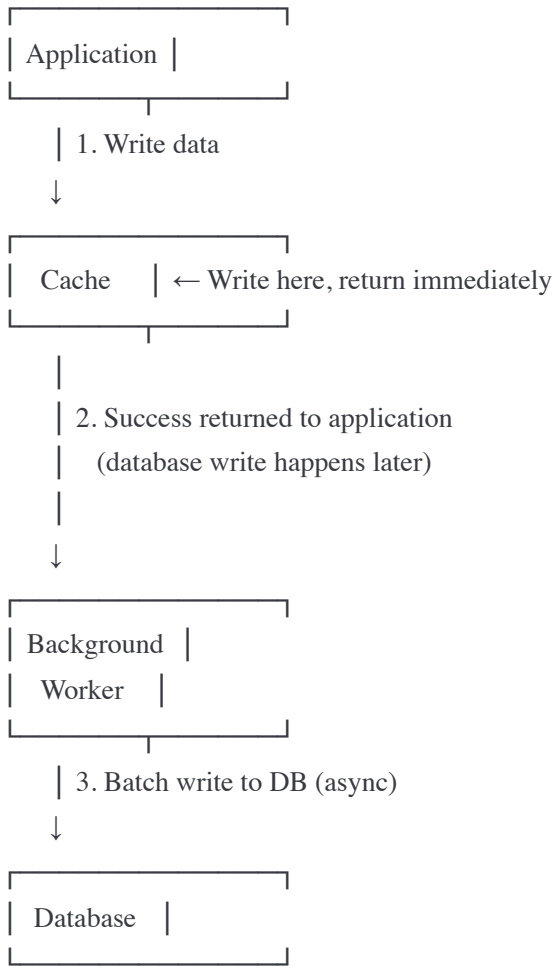
✗ DISADVANTAGES:
- Write latency higher (must update both cache and DB)
- Wasted cache space (rarely accessed items also cached)
- Cache failure breaks writes
- Extra write load on cache

---

3. Write-Back (Write-Behind) Cache

Writes go to cache first, database later. Fastest writes.

Write Flow:



Advantages:
- Super fast writes (cache is in memory)
- Can batch multiple writes
- Reduces database load

Risks:
- Data loss if cache crashes before DB write
- Complex to implement correctly

Python Implementation:

python

```
import threading
import queue
import time

class WriteBackCache:
    def __init__(self):
        self.cache = redis.Redis(host='localhost', port=6379, decode_responses=True)
        self.db = psycopg2.connect(...)

        # Queue for pending database writes
        self.write_queue = queue.Queue()

        # Start background worker
        self.worker_thread = threading.Thread(target=self._background_worker, daemon=True)
        self.worker_thread.start()

    def _background_worker(self):
        """Background thread that writes to database"""
        batch = []
        batch_size = 10
        batch_timeout = 5  # seconds
        last_flush = time.time()

        while True:
            try:
                # Get write operation (with timeout)
                operation = self.write_queue.get(timeout=1)
                batch.append(operation)

                # Flush batch if full or timeout reached
                if len(batch) >= batch_size or (time.time() - last_flush) > batch_timeout:
                    self._flush_batch(batch)
                    batch = []
                    last_flush = time.time()

            except queue.Empty:
                # Timeout - flush any pending writes
                if batch:
                    self._flush_batch(batch)
                    batch = []
                    last_flush = time.time()

    def _flush_batch(self, batch):
        """Write batch of operations to database"""
        if not batch:
            return
```

```
    try:
        cursor = self.db.cursor()

        for operation in batch:
            if operation['type'] == 'update':
                cursor.execute(
                    "UPDATE users SET name = %s, email = %s WHERE id = %s",
                    (operation['name'], operation['email'], operation['user_id'])
                )
            elif operation['type'] == 'create':
                cursor.execute(
                    "INSERT INTO users (id, name, email) VALUES (%s, %s, %s)",
                    (operation['user_id'], operation['name'], operation['email'])
                )

        self.db.commit()
        print(f"Flushed {len(batch)} operations to database")

    except Exception as e:
        self.db.rollback()
        print(f"Error flushing batch: {e}")
        # In production, would retry or alert

def update_user(self, user_id, data):
    """Write-back: Update cache immediately, queue DB write"""
    cache_key = f"user:{user_id}"

    # 1. Update cache immediately
    self.cache.setex(
        cache_key,
        3600,
        json.dumps({
            'id': user_id,
            'name': data['name'],
            'email': data['email']
        })
    )

    # 2. Queue database write (happens later)
    self.write_queue.put({
        'type': 'update',
        'user_id': user_id,
        'name': data['name'],
        'email': data['email']
    })
```

```
        # 3. Return immediately (FAST!)
        print(f"Updated cache for user {user_id} (DB write queued)")

    def get_user(self, user_id):
        """Read from cache (should always be there)"""
        cache_key = f"user:{user_id}"

        cached_user = self.cache.get(cache_key)
        if cached_user:
            return json.loads(cached_user)

        # Fallback to database if cache miss
        cursor = self.db.cursor()
        cursor.execute(
            "SELECT id, name, email FROM users WHERE id = %s",
            (user_id,)
        )
        row = cursor.fetchone()

        if row:
            return {'id': row[0], 'name': row[1], 'email': row[2]}

        return None

# Usage
write_back = WriteBackCache()

# Fast writes!
write_back.update_user(123, {'name': 'John', 'email': 'john@example.com'})
write_back.update_user(124, {'name': 'Jane', 'email': 'jane@example.com'})
write_back.update_user(125, {'name': 'Bob', 'email': 'bob@example.com'})
# All three return immediately!
# Database writes happen in background, batched together

time.sleep(6)  # Wait for background flush
```

Pros and Cons:

✓ ADVANTAGES:
- Extremely fast writes (memory speed)
- Can batch writes (reduces DB load)
- High write throughput
- Good for write-heavy workloads

✗ DISADVANTAGES:
- Risk of data loss (if cache crashes)
- Complex implementation
- Harder to debug
- Requires persistent cache (Redis with AOF/RDB)

When to use:
- Write-heavy workloads (logs, metrics, counters)
- Can tolerate small data loss
- Need maximum write performance

---

4. Refresh-Ahead Cache

Proactively refresh cache before expiration. Prevents cache misses.

Normal Cache Expiration:
Time 0: Load data into cache (TTL: 60 seconds)
Time 30: [Cache hit] - Return from cache
Time 60: Cache expires
Time 61: [Cache miss!] - User waits while data reloaded
Problem: User experiences slow request

Refresh-Ahead:
Time 0: Load data into cache (TTL: 60 seconds)
Time 30: [Cache hit] - Return from cache
Time 50: Background job refreshes cache (TTL approaching)
Time 60: Old cache expires, but NEW cache already loaded!
Time 61: [Cache hit] - User gets instant response
Solution: No cache miss, always fast!

Python Implementation:

python

```
import threading
import time
from datetime import datetime, timedelta

class RefreshAheadCache:
    def __init__(self):
        self.cache = {}
        self.metadata = {}  # Track last access, expiry
        self.lock = threading.Lock()

        # Start background refresh thread
        self.refresh_thread = threading.Thread(target=self._refresh_worker, daemon=True)
        self.refresh_thread.start()

    def get(self, key, fetch_function, ttl=60):
        """
        Get value from cache with refresh-ahead
        fetch_function: function to fetch data on miss
        ttl: time to live in seconds
        """
        with self.lock:
            now = datetime.now()

            # Check if in cache and not expired
            if key in self.cache:
                metadata = self.metadata[key]

                if now < metadata['expires_at']:
                    # Cache hit!
                    metadata['last_accessed'] = now
                    metadata['access_count'] += 1

                    # Check if should refresh ahead
                    time_until_expiry = (metadata['expires_at'] - now).total_seconds()
                    refresh_threshold = ttl * 0.2  # Refresh when 20% of TTL left

                    if time_until_expiry < refresh_threshold:
                        metadata['needs_refresh'] = True

                    return self.cache[key]

            # Cache miss - fetch data
            value = fetch_function(key)

            # Store in cache
            self.cache[key] = value
```

```
            self.metadata[key] = {
                'expires_at': now + timedelta(seconds=ttl),
                'last_accessed': now,
                'access_count': 1,
                'needs_refresh': False,
                'ttl': ttl,
                'fetch_function': fetch_function
            }

            return value

    def _refresh_worker(self):
        """Background thread that refreshes cache entries"""
        while True:
            time.sleep(5)  # Check every 5 seconds

            with self.lock:
                keys_to_refresh = [
                    key for key, meta in self.metadata.items()
                    if meta.get('needs_refresh', False) and meta['access_count'] > 2
                ]

            # Refresh popular entries that need it
            for key in keys_to_refresh:
                try:
                    with self.lock:
                        metadata = self.metadata[key]
                        fetch_function = metadata['fetch_function']
                        ttl = metadata['ttl']

                    # Fetch fresh data (outside lock)
                    new_value = fetch_function(key)

                    # Update cache
                    with self.lock:
                        self.cache[key] = new_value
                        self.metadata[key]['expires_at'] = datetime.now() + timedelta(seconds=ttl)
                        self.metadata[key]['needs_refresh'] = False

                    print(f"Refresh-ahead: Refreshed cache for key '{key}'")

                except Exception as e:
                    print(f"Error refreshing key '{key}': {e}")

# Example usage
def expensive_database_query(key):
    """Simulate expensive operation"""
```

```
    print(f"[DATABASE] Fetching data for {key}...")
    time.sleep(1)  # Simulate 1 second query
    return f"Data for {key} at {datetime.now()}"

refresh_cache = RefreshAheadCache()

# Simulate user accessing same data repeatedly
for i in range(20):
    data = refresh_cache.get('popular_item', expensive_database_query, ttl=10)
    print(f"Request {i+1}: {data}")
    time.sleep(2)  # User requests every 2 seconds

# Output shows:
# - First request: Fetches from database (1s)
# - Subsequent requests: Instant (cache hit)
# - Around 8 seconds: Background refresh happens
# - No user ever experiences cache miss!
```

Pros and Cons:

✓ ADVANTAGES:
- Eliminates cache miss latency for hot data
- Users always get fast responses
- Good for predictable access patterns
- Reduces database load spikes

✗ DISADVANTAGES:
- Wasted work if data not accessed again
- More complex implementation
- Requires accurate prediction of hot data
- Extra background processing

When to use:
- Frequently accessed data (hot keys)
- Predictable access patterns
- Can't tolerate cache miss latency
- Example: Homepage, popular products, trending posts

Strategy Comparison Table

Strategy	Read Speed	Write Spd	Consistency	Complexity	Best For
Cache-Aside	Fast	Fast	Eventual	Low	Read-heavy

(Lazy Load)	(on hit)				General use
Write-Through	Very Fast	Slow	Strong	Medium	Read-heavy
	(2 writes)			Need consist	
Write-Back (Write-Behind)	Very Fast	Very Fast (async)	Eventual (risky)	High	Write-heavy
				Can lose data	
Refresh-Ahead	Very Fast	N/A	Eventual	High	Hot data
	(no misses)			Predictable	

---

4. Cache Eviction Policies

When cache is full, which item do we remove?

1. LRU (Least Recently Used)

Remove the item that hasn't been used for the longest time.

Concept:
Items used recently are likely to be used again soon.
Items not used for a long time probably won't be needed.

Example:
Cache size: 3 items

Step 1: Add A
Cache: [A]

Step 2: Add B
Cache: [B, A] (B most recent)

Step 3: Add C
Cache: [C, B, A] (Cache full!)

Step 4: Access A (moves to front)
Cache: [A, C, B]

Step 5: Add D (cache full, evict B - least recently used)
Cache: [D, A, C]

Python Implementation:

python

```python
from collections import OrderedDict

class LRUCache:
    def __init__(self, capacity):
        self.cache = OrderedDict()
        self.capacity = capacity

    def get(self, key):
        """Get value and mark as recently used"""
        if key not in self.cache:
            return None

        # Move to end (most recent)
        self.cache.move_to_end(key)
        return self.cache[key]

    def put(self, key, value):
        """Add/update value"""
        if key in self.cache:
            # Update existing - move to end
            self.cache.move_to_end(key)
        else:
            # New item - check capacity
            if len(self.cache) >= self.capacity:
                # Remove least recently used (first item)
                oldest = next(iter(self.cache))
                print(f"Evicting: {oldest}")
                del self.cache[oldest]

        self.cache[key] = value

    def display(self):
        """Show cache contents (most recent last)"""
        print(f"Cache: {list(self.cache.keys())}")

# Example usage
cache = LRUCache(capacity=3)

print("Adding A, B, C:")
cache.put('A', 'Data A')
cache.put('B', 'Data B')
cache.put('C', 'Data C')
cache.display()  # [A, B, C]

print("\nAccessing A (makes it most recent):")
cache.get('A')
```

```
cache.display()  # [B, C, A]

print("\nAdding D (cache full, evicts B):")
cache.put('D', 'Data D')
cache.display()  # [C, A, D]

print("\nAccessing C:")
cache.get('C')
cache.display()  # [A, D, C]

print("\nAdding E (evicts A - least recently used):")
cache.put('E', 'Data E')
cache.display()  # [D, C, E]

# Output:
# Adding A, B, C:
# Cache: ['A', 'B', 'C']
#
# Accessing A (makes it most recent):
# Cache: ['B', 'C', 'A']
#
# Adding D (cache full, evicts B):
# Evicting: B
# Cache: ['C', 'A', 'D']
#
# Accessing C:
# Cache: ['A', 'D', 'C']
#
# Adding E (evicts A - least recently used):
# Evicting: A
# Cache: ['D', 'C', 'E']
```

When to use LRU:

✓ Good for:
- General purpose caching
- Temporal locality (recent items likely reused)
- Web page caching
- Most use cases

✗ Not good for:
- Sequential scans (everything gets accessed once)
- Batch processing

---

2. LFU (Least Frequently Used)

Remove the item used least often.

Concept:
Popular items should stay in cache.
Rarely used items should be evicted.

Example:
Cache size: 3 items

Add A (count: 1)
Add B (count: 1)
Add C (count: 1)
Cache: [A:1, B:1, C:1]

Access A (count: 2)
Access A (count: 3)
Access B (count: 2)
Cache: [A:3, B:2, C:1]

Add D (cache full, evict C - least frequently used)
Cache: [A:3, B:2, D:1]

Python Implementation:

```
python
```

```
from collections import defaultdict
import heapq

class LFUCache:
    def __init__(self, capacity):
        self.capacity = capacity
        self.cache = {}  # key -> value
        self.frequency = defaultdict(int)  # key -> frequency
        self.min_freq = 0
        self.freq_to_keys = defaultdict(set)  # frequency -> set of keys

    def get(self, key):
        """Get value and increment frequency"""
        if key not in self.cache:
            return None

        # Increment frequency
        self._increment_frequency(key)
        return self.cache[key]

    def put(self, key, value):
        """Add/update value"""
        if self.capacity <= 0:
            return

        if key in self.cache:
            # Update existing
            self.cache[key] = value
            self._increment_frequency(key)
            return

        # New item - check capacity
        if len(self.cache) >= self.capacity:
            # Evict least frequently used
            self._evict()

        # Add new item
        self.cache[key] = value
        self.frequency[key] = 1
        self.freq_to_keys[1].add(key)
        self.min_freq = 1

    def _increment_frequency(self, key):
        """Increase frequency count for key"""
        freq = self.frequency[key]
        self.frequency[key] = freq + 1
```

```
        # Remove from old frequency bucket
        self.freq_to_keys[freq].remove(key)
        if not self.freq_to_keys[freq]:
            del self.freq_to_keys[freq]
            if self.min_freq == freq:
                self.min_freq += 1

        # Add to new frequency bucket
        self.freq_to_keys[freq + 1].add(key)

    def _evict(self):
        """Remove least frequently used item"""
        # Get any key from min frequency bucket
        keys_with_min_freq = self.freq_to_keys[self.min_freq]
        key_to_evict = keys_with_min_freq.pop()

        if not keys_with_min_freq:
            del self.freq_to_keys[self.min_freq]

        # Remove from cache
        del self.cache[key_to_evict]
        del self.frequency[key_to_evict]

        print(f"Evicting: {key_to_evict} (frequency: {self.min_freq})")

    def display(self):
        """Show cache with frequencies"""
        items = [(k, f"freq={self.frequency[k]}") for k in self.cache.keys()]
        print(f"Cache: {items}")

# Example usage
cache = LFUCache(capacity=3)

print("Adding A, B, C:")
cache.put('A', 'Data A')
cache.put('B', 'Data B')
cache.put('C', 'Data C')
cache.display()

print("\nAccessing A three times, B once:")
cache.get('A')
cache.get('A')
cache.get('A')
cache.get('B')
cache.display()
```

```
print("\nAdding D (evicts C - least frequently used):")
cache.put('D', 'Data D')
cache.display()

print("\nAdding E (evicts D - least frequently used):")
cache.put('E', 'Data E')
cache.display()

# Output:
# Adding A, B, C:
# Cache: [('A', 'freq=1'), ('B', 'freq=1'), ('C', 'freq=1')]
#
# Accessing A three times, B once:
# Cache: [('A', 'freq=4'), ('B', 'freq=2'), ('C', 'freq=1')]
#
# Adding D (evicts C - least frequently used):
# Evicting: C (frequency: 1)
# Cache: [('A', 'freq=4'), ('B', 'freq=2'), ('D', 'freq=1')]
#
# Adding E (evicts D - least frequently used):
# Evicting: D (frequency: 1)
# Cache: [('A', 'freq=4'), ('B', 'freq=2'), ('E', 'freq=1')]
```

When to use LFU:

✓ Good for:
- Frequency-based access patterns
- Popular items accessed repeatedly
- Content recommendation systems

✗ Not good for:
- Access patterns change over time
- New popular items can't compete with old popular items

---

3. FIFO (First In, First Out)

Remove the oldest item in cache.

Concept:
Simple queue - oldest items get evicted first.

Example:
Cache size: 3 items

Add A → Cache: [A]
Add B → Cache: [A, B]
Add C → Cache: [A, B, C] (full)
Add D → Cache: [B, C, D] (A evicted - first in)
Add E → Cache: [C, D, E] (B evicted)

Python Implementation:

python

```python
from collections import deque

class FIFOCache:
    def __init__(self, capacity):
        self.capacity = capacity
        self.cache = {}  # key -> value
        self.order = deque()  # track insertion order

    def get(self, key):
        """Get value (doesn't affect eviction order)"""
        return self.cache.get(key)

    def put(self, key, value):
        """Add/update value"""
        if key in self.cache:
            # Update existing (keep same position)
            self.cache[key] = value
            return

        # New item - check capacity
        if len(self.cache) >= self.capacity:
            # Evict oldest (first in queue)
            oldest = self.order.popleft()
            print(f"Evicting: {oldest}")
            del self.cache[oldest]

        # Add new item
        self.cache[key] = value
        self.order.append(key)

    def display(self):
        """Show cache (left = oldest, right = newest)"""
        print(f"Cache: {list(self.order)}")

# Example usage
cache = FIFOCache(capacity=3)

cache.put('A', 1)
cache.put('B', 2)
cache.put('C', 3)
cache.display()  # [A, B, C]

cache.get('A')  # Access A (doesn't change order in FIFO!)
cache.display()  # [A, B, C]  (same)

cache.put('D', 4)  # Evicts A (oldest)
```

```
cache.display()  # [B, C, D]

# Output:
# Cache: ['A', 'B', 'C']
# Cache: ['A', 'B', 'C']
# Evicting: A
# Cache: ['B', 'C', 'D']
```

When to use FIFO:

✓ Good for:
- Simple implementation needed
- All items equally important
- No clear access pattern

✗ Not good for:
- Frequently accessed items (may get evicted!)
- Most real-world scenarios (LRU is usually better)

---

4. Random Replacement

Randomly pick item to evict.

python

```
import random

class RandomCache:
    def __init__(self, capacity):
        self.capacity = capacity
        self.cache = {}

    def get(self, key):
        return self.cache.get(key)

    def put(self, key, value):
        if key in self.cache:
            self.cache[key] = value
            return

        if len(self.cache) >= self.capacity:
            # Randomly evict
            victim = random.choice(list(self.cache.keys()))
            print(f"Randomly evicting: {victim}")
            del self.cache[victim]

        self.cache[key] = value
```

When to use Random:

✓ Good for:
- Extremely simple implementation
- Uniform access patterns
- When you really don't know the pattern

✗ Not good for:
- Most scenarios (other policies perform better)

Eviction Policy Comparison

Policy	Complexity	Hit Ratio	Best For	Worst For
LRU	O(1)	High	General use Temporal locality	Sequential scan
LFU	O(1)	High	Popular items Frequency	Changing access patterns

			based	
FIFO	O(1)	Low	Simple needs	Hot items
			Equal items	
Random	O(1)	Low	Uniform	Most scenarios
			access	

Recommendation: Start with LRU (best general-purpose)

---

5. Cache Invalidation Challenges

> "There are only two hard things in Computer Science: cache invalidation and naming things." - Phil Karlton

The Problem

Scenario:
1. User reads profile (cached)
2. User updates profile
3. Now cache has OLD data, database has NEW data
4. Other users see OLD profile from cache!

This is CACHE INVALIDATION problem.

Challenge 1: Stale Data

Problem: Cache contains outdated information.

Timeline:
00:00 - User views product price: $100 (cached)
00:05 - Admin updates price to $80 in database
00:10 - User views product again
    Still sees $100 from cache! (WRONG)
00:15 - Cache expires (TTL), fresh data loaded
    Now sees correct $80

Problem: 15 minutes of wrong prices!

Solutions:

python

```python
# Solution 1: Short TTL
cache.setex('product:123', 60, json.dumps(product))  # 1 minute TTL
# Pros: Simple
# Cons: Still some staleness, more DB load

# Solution 2: Explicit Invalidation
def update_product(product_id, new_data):
    # Update database
    db.update(product_id, new_data)

    # Invalidate cache
    cache.delete(f'product:{product_id}')
# Pros: No stale data
# Cons: Must remember to invalidate everywhere

# Solution 3: Write-Through
def update_product(product_id, new_data):
    # Update cache
    cache.set(f'product:{product_id}', new_data)

    # Update database
    db.update(product_id, new_data)
# Pros: Always consistent
# Cons: Slower writes

# Solution 4: Versioning
product = {
    'id': 123,
    'price': 100,
    'version': 5  # Increment on every update
}

def get_product(product_id):
    cached = cache.get(f'product:{product_id}')
    db_version = db.get_version(product_id)

    if cached and cached['version'] == db_version:
        return cached  # Valid

    # Stale or missing - fetch fresh
    fresh = db.get(product_id)
    cache.set(f'product:{product_id}', fresh)
    return fresh
```

Challenge 2: Thundering Herd

Problem: Many requests for expired cache item hit database simultaneously.

Scenario:
Popular item cached with 1-hour TTL
100,000 users viewing it

Timeline:
Hour 1: All requests hit cache (fast)
Hour 1 + 1 second: Cache expires
Next 5 seconds: 10,000 requests arrive
All 10,000 requests:
- Check cache (miss)
- Query database (10,000 simultaneous queries!)
- Database crashes! 💥

This is THUNDERING HERD problem.

Solutions:

python

```python
import threading
import time

# Solution 1: Mutex/Lock (only one request fetches)
class ThunderingHerdProtection:
    def __init__(self):
        self.cache = {}
        self.locks = {}
        self.global_lock = threading.Lock()

    def get(self, key, fetch_function):
        # Check cache
        if key in self.cache:
            return self.cache[key]

        # Get or create lock for this key
        with self.global_lock:
            if key not in self.locks:
                self.locks[key] = threading.Lock()
            lock = self.locks[key]

        # Only one thread fetches, others wait
        with lock:
            # Double-check cache (another thread might have loaded it)
            if key in self.cache:
                return self.cache[key]

            # Fetch data (only one thread does this!)
            print(f"[{threading.current_thread().name}] Fetching {key} from database")
            value = fetch_function(key)

            # Store in cache
            self.cache[key] = value
            return value

# Test thundering herd protection
def slow_fetch(key):
    time.sleep(1)  # Simulate slow database
    return f"Data for {key}"

cache = ThunderingHerdProtection()

# Simulate 10 simultaneous requests
threads = []
for i in range(10):
    t = threading.Thread(
```

```
        target=lambda: cache.get('popular_item', slow_fetch),
        name=f"Thread-{i}"
    )
    threads.append(t)
    t.start()

for t in threads:
    t.join()

# Output:
# [Thread-0] Fetching popular_item from database
# (Only ONE thread fetches, others wait and reuse result!)

# Solution 2: Probabilistic Early Expiration
import random

def get_with_early_expiration(key, fetch_function, ttl=3600):
    cached = cache.get(key)

    if cached:
        # Check if should refresh early (probabilistically)
        time_left = cached['expires_at'] - time.time()

        # Chance of early refresh increases as expiration approaches
        refresh_probability = 1 - (time_left / ttl)

        if random.random() < refresh_probability:
            # Refresh now (proactively)
            background_refresh(key, fetch_function)

        return cached['value']

    # Cache miss
    value = fetch_function(key)
    cache.set(key, {'value': value, 'expires_at': time.time() + ttl})
    return value

# Solution 3: Refresh-Ahead (covered earlier)
# Proactively refresh before expiration
```

Challenge 3: Cache Penetration

Problem: Requests for non-existent data bypass cache, hitting database.

Attack Scenario:

Attacker requests product:99999999 (doesn't exist)

→ Cache miss

→ Database query (not found)

→ Return 404

Attacker repeats 10,000 times:

→ 10,000 cache misses

→ 10,000 database queries

→ Database overloaded!

Cache is useless because item doesn't exist!

Solutions:

python

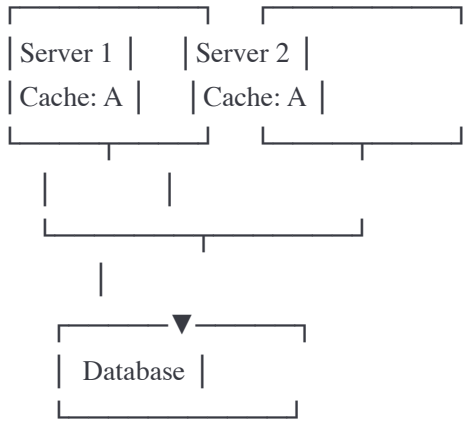
```python
# Solution 1: Cache Negative Results
def get_user_safe(user_id):
    # Check cache
    cached = cache.get(f'user:{user_id}')

    if cached == 'NOT_FOUND':
        return None  # Cached negative result

    if cached:
        return cached

    # Fetch from database
    user = db.get_user(user_id)

    if user:
        cache.setex(f'user:{user_id}', 3600, json.dumps(user))
        return user
    else:
        # Cache the fact that user doesn't exist!
        cache.setex(f'user:{user_id}', 300, 'NOT_FOUND')  # Short TTL
        return None

# Solution 2: Bloom Filter (space-efficient check)
from pybloom_live import BloomFilter

# Create bloom filter with all existing keys
valid_keys = BloomFilter(capacity=1000000, error_rate=0.001)

# Add all valid product IDs
for product_id in db.get_all_product_ids():
    valid_keys.add(product_id)

def get_product_safe(product_id):
    # Quick check: does this ID exist?
    if product_id not in valid_keys:
        # Definitely doesn't exist (no DB query needed!)
        return None

    # Might exist (small false positive rate)
    # Proceed with normal cache logic
    return get_product_with_cache(product_id)

# Bloom filter benefits:
# - Very small memory footprint (1MB for 1M items)
```

```
# - O(1) lookup
# - Protects database from non-existent key queries
```

Challenge 4: Cache Consistency Across Instances

Problem: Multiple cache instances can have different data.

Multi-Server Setup:



Problem:

1. User updates data via Server 1
2. Server 1 invalidates its cache
3. Server 2 still has old cached data!
4. Users hitting Server 2 see stale data

Solution: Shared cache or cache invalidation messages

Solutions:

python

```python
# Solution 1: Use Centralized Cache (Redis)
# All servers share same Redis instance
# No consistency issues!

import redis

redis_cache = redis.Redis(host='cache-server', port=6379)

def update_user(user_id, data):
    # Update database
    db.update(user_id, data)

    # Invalidate in shared cache
    redis_cache.delete(f'user:{user_id}')
    # All servers immediately see the invalidation!

# Solution 2: Cache Invalidation Messages
import pika  # RabbitMQ

def update_user_with_broadcast(user_id, data):
    # Update database
    db.update(user_id, data)

    # Broadcast invalidation message
    connection = pika.BlockingConnection()
    channel = connection.channel()
    channel.basic_publish(
        exchange='cache_invalidation',
        routing_key='',
        body=json.dumps({'action': 'invalidate', 'key': f'user:{user_id}'})
    )
    # All servers receive message and invalidate their local caches

# Each server listens for invalidation messages:
def listen_for_invalidations():
    connection = pika.BlockingConnection()
    channel = connection.channel()
    channel.exchange_declare(exchange='cache_invalidation', exchange_type='fanout')

    def callback(ch, method, properties, body):
        message = json.loads(body)
        if message['action'] == 'invalidate':
            local_cache.delete(message['key'])
            print(f"Invalidated {message['key']}")
```

```
channel.basic_consume(queue='cache_updates', on_message_callback=callback)
channel.start_consuming()
```

---

6. Distributed Caching: Redis & Memcached

Redis

In-memory data structure store. Supports rich data types.

Features:

- Key-value store
- Data structures: Strings, Lists, Sets, Hashes, Sorted Sets
- Persistence (optional)
- Replication
- Pub/Sub
- Lua scripting

Python with Redis:

```python
```

```python
import redis
import json

# Connect to Redis
r = redis.Redis(host='localhost', port=6379, decode_responses=True)

# 1. Simple key-value
r.set('user:1000', json.dumps({'name': 'John', 'age': 30}))
user = json.loads(r.get('user:1000'))

# 2. With expiration (TTL)
r.setex('session:abc123', 3600, 'session_data')  # Expires in 1 hour

# 3. Increment counter (atomic)
r.incr('page:views')  # Increments by 1
views = r.get('page:views')

# 4. Hash (object-like)
r.hset('user:1001', mapping={
    'name': 'Jane',
    'email': 'jane@example.com',
    'age': '25'
})
name = r.hget('user:1001', 'name')
all_fields = r.hgetall('user:1001')

# 5. List (queue)
r.lpush('queue:tasks', 'task1', 'task2', 'task3')  # Push left
task = r.rpop('queue:tasks')  # Pop right (FIFO)

# 6. Set (unique items)
r.sadd('user:1000:followers', '1001', '1002', '1003')
followers = r.smembers('user:1000:followers')
is_follower = r.sismember('user:1000:followers', '1001')

# 7. Sorted Set (leaderboard)
r.zadd('leaderboard', {'player1': 100, 'player2': 200, 'player3': 150})
top_players = r.zrevrange('leaderboard', 0, 9, withscores=True)  # Top 10

# 8. Pub/Sub
def message_handler(message):
    print(f"Received: {message['data']}")

pubsub = r.pubsub()
pubsub.subscribe(**{'notifications': message_handler})
pubsub.run_in_thread(sleep_time=0.1)
```

```
r.publish('notifications', 'Hello, world!')

# 9. Transactions
pipe = r.pipeline()
pipe.set('key1', 'value1')
pipe.set('key2', 'value2')
pipe.incr('counter')
pipe.execute()  # All or nothing

# 10. Lua script (atomic operations)
lua_script = """
local current = redis.call('get', KEYS[1])
if current == ARGV[1] then
    return redis.call('set', KEYS[1], ARGV[2])
else
    return 0
end
"""
# Compare and swap
result = r.eval(lua_script, 1, 'mykey', 'old_value', 'new_value')
```

Redis Configuration for Production:

```
# redis.conf

# Memory
maxmemory 2gb
maxmemory-policy allkeys-lru  # Eviction policy

# Persistence (optional - for durability)
save 900 1      # Save if 1 key changed in 15 min
save 300 10     # Save if 10 keys changed in 5 min
save 60 10000   # Save if 10000 keys changed in 1 min

# AOF (Append-Only File) - more durable
appendonly yes
appendfsync everysec

# Replication (for high availability)
# On replica:
replicaof master-ip 6379

# Security
requirepass your_password_here

# Performance
tcp-backlog 511
timeout 0
tcp-keepalive 300
```

Memcached

Simple, high-performance key-value cache. Distributed by default.

Features:

- Very fast (multi-threaded)
- Simple (only strings)
- No persistence
- Distributed hash table
- Built-in consistent hashing

Python with Memcached:

```
python
```

```python
from pymemcache.client import base

# Connect to Memcached
client = base.Client(('localhost', 11211))

# 1. Set value
client.set('user:1000', json.dumps({'name': 'John', 'age': 30}))

# 2. Get value
user = json.loads(client.get('user:1000'))

# 3. Set with expiration
client.set('session:abc', 'data', expire=3600)  # 1 hour

# 4. Add (only if doesn't exist)
client.add('key', 'value')

# 5. Replace (only if exists)
client.replace('key', 'new_value')

# 6. Delete
client.delete('key')

# 7. Increment/Decrement
client.set('counter', '0')
client.incr('counter', 1)  # Increment by 1
client.decr('counter', 1)  # Decrement by 1

# 8. Multi-get (efficient batch operation)
results = client.get_many(['key1', 'key2', 'key3'])

# 9. Multi-set
client.set_many({
    'key1': 'value1',
    'key2': 'value2',
    'key3': 'value3'
}, expire=3600)

# 10. Stats
stats = client.stats()
print(f"Total items: {stats[b'curr_items']}")
print(f"Hits: {stats[b'get_hits']}")
print(f"Misses: {stats[b'get_misses']}")
```

Distributed Memcached (Multiple Servers):

```python
from pymemcache.client.hash import HashClient

# Multiple Memcached servers
servers = [
    ('cache1.example.com', 11211),
    ('cache2.example.com', 11211),
    ('cache3.example.com', 11211)
]

# Client automatically distributes keys across servers
client = HashClient(servers)

# Keys are hashed and distributed
client.set('user:1', 'data1')  # Might go to cache1
client.set('user:2', 'data2')  # Might go to cache2
client.set('user:3', 'data3')  # Might go to cache3

# Client handles consistent hashing automatically
```

Redis vs Memcached

Feature	Redis	Memcached
Data Types	Rich	String only
Persistence	Yes	No
Replication	Yes	No
Transactions	Yes	No
Pub/Sub	Yes	No
Lua Scripts	Yes	No
Performance	Very fast	Faster
Threading	Single	Multi
Memory Efficiency	Good	Better
Use Case	Complex	Simple
	Session	Page cache
	Leaderboard	DB query
	Real-time	cache

General Recommendation:

- Use Redis (more features, same great performance)
- Use Memcached only if you need absolute max throughput for simple caching

---

7. CDN (Content Delivery Networks)

CDN: Geographically distributed servers that cache content close to users.

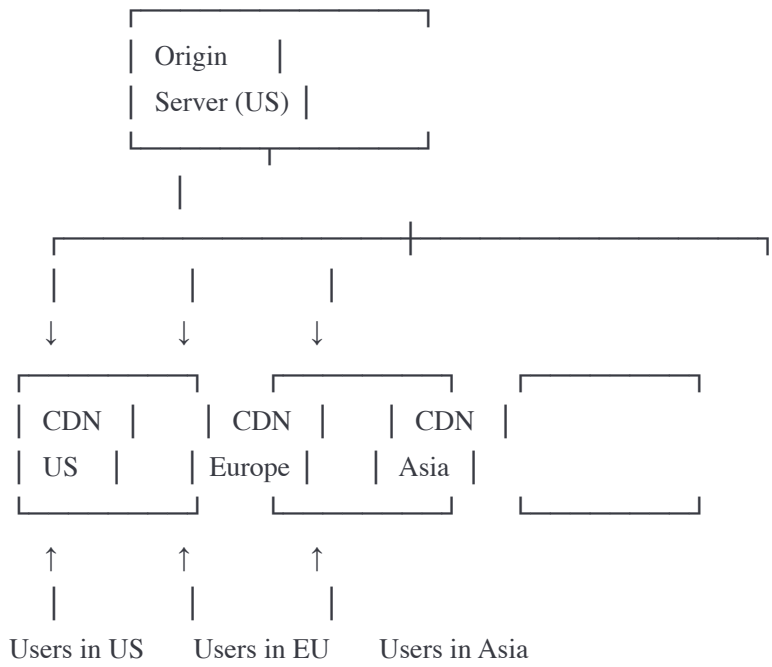
How CDN Works

Without CDN:
User in Tokyo → Request → Server in US (8,000 km away)
Latency: 150ms

With CDN:
User in Tokyo → Request → CDN in Tokyo (nearby)
Latency: 5ms

30x faster!

Architecture:



CDN Benefits

1. REDUCED LATENCY
   Content served from nearby server
   Example: 150ms → 5ms (30x faster)

2. REDUCED BANDWIDTH COSTS
   Origin server serves content once

CDN serves millions of requests

3. IMPROVED AVAILABILITY
   If origin down, CDN can still serve cached content

4. DDoS PROTECTION
   CDN absorbs attack traffic
   Origin server protected

5. OFFLOAD TRAFFIC
   Origin server only handles cache misses
   Example: 99% of requests hit CDN, 1% hit origin

CDN Workflow

First Request (Cold Cache):
1. User in Tokyo requests image.jpg
2. Request goes to Tokyo CDN
3. CDN doesn't have it (cache miss)
4. CDN fetches from origin server
5. CDN caches image.jpg
6. CDN returns to user
   Time: ~150ms (first time penalty)

Subsequent Requests (Warm Cache):
1. User in Tokyo requests image.jpg
2. Request goes to Tokyo CDN
3. CDN has it (cache hit!)
4. CDN returns immediately
   Time: ~5ms (super fast!)

Other users in Tokyo:
- Also get image.jpg from Tokyo CDN (fast!)

Users in other regions:
- Sydney CDN caches it for Sydney users
- Mumbai CDN caches it for Mumbai users
- etc.

CDN Configuration Example (CloudFront)

javascript

```
// CloudFront distribution configuration
{
  "Origins": [{
    "Id": "my-origin",
    "DomainName": "example.com",
    "CustomHeaders": [{
      "HeaderName": "X-Custom-Header",
      "HeaderValue": "value"
    }]
  }],

  "DefaultCacheBehavior": {
    "TargetOriginId": "my-origin",
    "ViewerProtocolPolicy": "redirect-to-https",

    // Cache settings
    "MinTTL": 0,
    "DefaultTTL": 86400,      // 24 hours
    "MaxTTL": 31536000,       // 1 year

    // What to cache based on
    "ForwardedValues": {
      "QueryString": false,   // Don't cache based on query params
      "Cookies": {
        "Forward": "none"     // Don't cache based on cookies
      },
      "Headers": []           // Don't cache based on headers
    },

    // Compression
    "Compress": true,         // Auto gzip/brotli compression

    // Allowed HTTP methods
    "AllowedMethods": ["GET", "HEAD", "OPTIONS"],
    "CachedMethods": ["GET", "HEAD"]
  },

  // Custom error responses
  "CustomErrorResponses": [{
    "ErrorCode": 404,
    "ResponseCode": 404,
    "ResponsePagePath": "/404.html",
    "ErrorCachingMinTTL": 300
  }]
}
```

CDN Cache Control Headers

http

```
# Control how CDN caches content

# Cache for 1 year (static assets)
Cache-Control: public, max-age=31536000, immutable

# Cache for 1 hour
Cache-Control: public, max-age=3600

# Don't cache (dynamic content)
Cache-Control: no-store, no-cache, must-revalidate

# Cache but revalidate (API responses)
Cache-Control: public, max-age=0, must-revalidate

# Cache Control Directives:
# public        - Can be cached by any cache
# private       - Only cached by browser (not CDN)
# no-cache      - Must revalidate before use
# no-store      - Don't cache at all
# max-age       - Cache lifetime in seconds
# s-maxage      - CDN cache lifetime (overrides max-age)
# immutable     - Never needs revalidation
# stale-while-revalidate - Serve stale while fetching fresh
```

Example - Setting Headers:

python

```python
from flask import Flask, send_file, make_response

app = Flask(__name__)

@app.route('/static/<path:filename>')
def serve_static(filename):
    """Serve static files with long cache"""
    response = make_response(send_file(f'static/{filename}'))

    # Cache for 1 year
    response.headers['Cache-Control'] = 'public, max-age=31536000, immutable'

    return response

@app.route('/api/data')
def serve_api():
    """Serve API with short cache"""
    response = make_response(get_api_data())

    # Cache for 5 minutes, but CDN can serve stale for 1 hour while refreshing
    response.headers['Cache-Control'] = 'public, max-age=300, stale-while-revalidate=3600'

    return response

@app.route('/user/profile')
def user_profile():
    """User-specific data - don't cache on CDN"""
    response = make_response(get_user_profile())

    # Browser can cache, but not CDN
    response.headers['Cache-Control'] = 'private, max-age=300'

    return response
```

CDN Invalidation

Problem: Cached content is stale, need to update it.

Methods:

1. WAIT FOR TTL
   Let cache expire naturally
   Simple but slow

2. VERSIONED URLS
   Change URL when content changes
   /assets/app.v1.js → /assets/app.v2.js
   Best practice!

3. INVALIDATION API
   Tell CDN to purge specific files
   Fast but costs money

4. WILDCARD INVALIDATION
   Purge all files matching pattern
   /images/* → purges all images

Example - CloudFront Invalidation:

python

```python
import boto3

def invalidate_cdn(paths):
    """Invalidate CloudFront cache"""
    client = boto3.client('cloudfront')

    response = client.create_invalidation(
        DistributionId='E1234567890ABC',
        InvalidationBatch={
            'Paths': {
                'Quantity': len(paths),
                'Items': paths
            },
            'CallerReference': str(time.time())
        }
    )

    print(f"Invalidation created: {response['Invalidation']['Id']}")

# Usage
invalidate_cdn([
    '/index.html',
    '/css/*',          # All CSS files
    '/images/logo.png'
])
```

Key Takeaways

1. **Caching Basics:**
   - Store frequently accessed data in fast storage
   - Can improve performance 100-1000x
   - Use when read-heavy, expensive to compute, or slow to fetch
2. **Cache Strategies:**
   - Cache-Aside: Most common, app manages cache
   - Write-Through: Always consistent, slower writes
   - Write-Back: Fast writes, risk of data loss
   - Refresh-Ahead: Proactive refresh for hot data
3. **Eviction Policies:**
   - LRU: Best general-purpose (evict least recently used)

- LFU: Good for popularity-based (evict least frequently used)
- FIFO: Simple (evict oldest)
- Choose based on access pattern

4. **Invalidation:**
   - Hardest problem in caching
   - Use short TTL, explicit invalidation, or versioned URLs
   - Watch for thundering herd and cache penetration
5. **Distributed Caching:**
   - Redis: Feature-rich, persistence, pub/sub
   - Memcached: Simple, very fast, distributed
   - Use Redis for most cases
6. **CDN:**
   - Caches content geographically close to users
   - Reduces latency 10-100x for global users
   - Essential for static assets (images, CSS, JS)
   - Use Cache-Control headers effectively

Practice Problems

1. Design a caching layer for a news website with 1 million daily visitors.
2. Implement an LRU cache from scratch with O(1) get and put operations.
3. How would you handle cache invalidation for a social media feed?

Next Chapter Preview

In Chapter 6, we'll explore **Database Fundamentals**:

- SQL vs NoSQL
- ACID properties
- Database indexes
- Normalization vs denormalization
- Connection pooling

Ready to continue?