**Transparent Hierarchical Cache: Comprehensive Architecture and Design Principles**

**Overview**

The Transparent Hierarchical Cache is a sophisticated multi-tier caching system designed to provide high-performance data access through intelligent layering and advanced parameter-based searching capabilities. It implements a **transparent** approach where the complexity of multiple cache layers is hidden from the application, presenting a unified interface while optimizing performance through strategic data placement and retrieval.

**Core Architecture Principles**

**1. Multi-Tier Hierarchy (L1 → L2 → L3)**

The system implements a **three-level hierarchy** with distinct characteristics:

* **L1 (Local Cache)**: In-memory, process-local cache using Caffeine
  + Fastest access (nanosecond latency)
  + Limited by JVM memory
  + No network overhead
  + Process-specific data
* **L2 (Redis Cache)**: Distributed, shared remote cache
  + Fast network access (microsecond latency)
  + Shared across application instances
  + Volatile but persistent across restarts
  + High throughput, low latency
* **L3 (Database Cache)**: Persistent, searchable storage
  + Slower access (millisecond latency)
  + Fully persistent and durable
  + Complex query capabilities
  + Ultimate source of truth

**2. Transparency Principle**

The cache operates **transparently** by:

* **Unified API**: Single interface regardless of which layer serves the data
* **Automatic Fallback**: Seamless progression through cache layers
* **Invisible Complexity**: Application code doesn't need to know about the hierarchy
* **Consistent Behaviour**: Same semantics across all layers

**Advanced Parameter-Based Caching System**

**3. Hierarchical Search Parameters**

The most sophisticated aspect is the **multilevel search parameter** system:

**SearchParameter Structure**

SearchParameter {  
 name: String // Parameter name (e.g., "category", "status")  
 value: String // Parameter value (e.g., "electronics", "active")  
 level: int // Hierarchy level (0, 1, 2, ...)  
}

**Hierarchical Pattern Generation**

The system generates **hierarchical patterns** from parameter lists:

**Example**: Parameters [L1:category=Books, L2:status=ACTIVE, L3:region=US]

**Generated Patterns**:

Individual patterns:  
- L1:category=Books  
- L2:status=ACTIVE   
- L3:region=US  
  
Hierarchical combinations:  
- L1:category=Books>L2:status=ACTIVE  
- L2:status=ACTIVE>L3:region=US  
- L1:category=Books>L2:status=ACTIVE>L3:region=US  
  
All possible sub-hierarchies:  
- L1:category=Books>L3:region=US  
- (and all other valid combinations maintaining level order)

**4. Multi-Level Indexing Strategy**

**Local Parameter Cache (L1)**

localParamCache: Map<String, Set<String>>  
Key: Pattern string (e.g., "L1:category=Books>L2:status=ACTIVE")  
Value: Set of unique IDs that match this pattern

**Database Parameter Index (L3)**

cache\_param\_index {  
 param\_pattern: VARCHAR(*1000*) -- Pattern string  
 unique\_string\_id: VARCHAR(*500*) -- Reference to cached item  
}

**Pattern Matching Logic**

**Storage Process**:

When storing an item with parameters [category=Books, status=ACTIVE]

1. Generate all hierarchical patterns
2. For each pattern, add the item's unique ID to the pattern's ID set
3. Store in both local cache and database index

**Search Process**:

1. Convert search parameters to patterns
2. Query pattern indexes to get candidate unique IDs
3. Retrieve items by unique IDs
4. Apply additional filtering if needed

**5. Intelligent Search Degradation**

The system supports **flexible parameter matching**:

**Exact Match First**

* Try exact pattern match: L1:category=Books>L2:status=ACTIVE

**Hierarchical Degradation**

If exact match fails, find items with **more specific patterns**:

* Item stored with: L1:category=Books>L2:status=ACTIVE>L3:region=US
* Search for: L1:category=Books>L2:status=ACTIVE
* **Match Found**: The stored item contains the search criteria

**Partial Match Strategy**

* Search can match items that have **additional parameters** beyond the search criteria
* Enables finding "more specific" cached results for "less specific" queries

**Cache Addressing and Key Management**

**6. Multi-Modal Key System**

Items can be addressed through **multiple key types**:

**Primary Addressing**

* **String Key**: Human-readable identifier ("user:profile:123")
* **Long ID**: Numeric identifier (123456L)
* **Composite Key**: Combination ("user:profile" + 123456L)

**Unique ID Generation**

**uniqueId = stringKey + (longId != null ? ":" + longId : "")**// Examples:  
// "user:profile" (string key only)  
// "user:profile:123" (string + long)

**Multi-Index Lookups**

* **Primary Cache**: uniqueId → CachedItem
* **Long Key Cache**: longId → uniqueId (reverse lookup)
* **Parameter Cache**: pattern → Set<uniqueId>

**7. Linking and Association System**

The system supports **dynamic linking** of cache entries:

**Key-ID Linking**

// Store item with string key only  
put("user:profile", parameters, userData);  
  
// Later, associate with numeric ID  
link("user:profile", 123456L);  
// Now accessible by both "user:profile" and 123456L

**Parameter Linking**

// Add additional search parameters to existing cached item  
link("user:profile", newParameters);  
// Item now discoverable through additional parameter combinations

**Fallback Strategy System**

**8. Configurable Fallback Strategies**

The system implements **intelligent fallback** with multiple strategies:

**REDIS\_THEN\_DATABASE**

L1 miss → L2 (Redis) → L3 (Database) → Compute

* Optimizes for Redis performance
* Database as final fallback

**DATABASE\_THEN\_REDIS**

L1 miss → L3 (Database) → L2 (Redis) → Compute

* Database as authoritative source
* Redis for acceleration

**Layer-Specific Strategies**

* : Skip database entirely **REDIS\_ONLY**
* : Skip Redis entirely **DATABASE\_ONLY**

**9. Write-Through and Read-Through Patterns**

**Write-Through Strategy**

**put**(key, parameters, value) {  
 // Always write to L1  
 writeToLocalCache(key, parameters, value);  
   
 // Conditionally write to remote layers  
 if (writeThrough) {  
 writeToRedis(key, parameters, value);  
 writeToDatabase(key, parameters, value);  
 }  
}

**Read-Through Strategy**

**get**(key) {  
 // Check L1 first  
 if (L1.contains(key)) return L1.get(key);  
   
 // Apply fallback strategy  
 value = applyFallbackStrategy(key);  
   
 // Backfill L1  
 if (value != null) L1.put(key, value);  
 return value;  
}

**Advanced Features**

**10. Context-Aware Operations**

The system supports **runtime context** for specialized behaviour:

CacheContext.set(  
 CacheContext.builder()  
 .skipLocalCache(true) // Force remote lookup  
 .forceRefresh(true) // Bypass all caches  
 .fallbackStrategy(REDIS\_ONLY) // Override global strategy  
 .customTtl(300000L) // Custom TTL for this operation  
 .build()  
);

**11. Comprehensive Statistics**

**Multi-Level Statistics Tracking**:

* **L1 Statistics**: Hits, misses, puts, evictions
* **L2 Statistics**: Hits, misses, puts, errors
* **L3 Statistics**: Hits, misses, puts, errors
* **Overall Statistics**: Combined metrics and hit rates

**12. Intelligent Invalidation**

**Cascading Invalidation**

**invalidate**(key) {  
 // Find all related identifiers  
 findAssociatedIds(key);  
   
 // Invalidate across all layers  
 invalidateL1(key, id, parameters);  
 invalidateL2(key, id);  
 invalidateL3(key, id);  
   
 // Clean up parameter indexes  
 removeFromParameterIndexes(parameters);  
}

**Pattern-Based Invalidation**

* Can invalidate by parameter patterns
* Removes all items matching specific parameter combinations

**Performance Optimization Principles**

**13. Memory Management**

* **Bounded Local Caches**: Configurable size limits prevent memory exhaustion
* **Parameter Cache Sizing**: Separate size controls for parameter indexes
* **Automatic Eviction**: LRU-based eviction with TTL support

**14. Network Optimization**

* **Batch Operations**: Multiple operations combined into single network calls
* **Async Writes**: Non-blocking writes to remote layers
* **Connection Pooling**: Efficient connection management for database operations

**15. Serialization Strategy**

* **Thread-Local Serializers**: Kryo serializers per thread to avoid contention
* **Binary Storage**: Efficient binary representation in database
* **Class Registration**: Pre-registered classes for faster serialization

**Use Case Applications**

**16. Ideal Scenarios**

**Complex Query Caching**

* Cache results of expensive database queries
* Enable parameter-based search without re-executing queries
* Support partial matching for query optimization

**Multi-Tenant Applications**

* Cache tenant-specific data with parameter isolation
* Enable cross-tenant pattern matching where appropriate
* Maintain tenant-specific TTLs and policies

**API Response Caching**

* Cache API responses with multiple access patterns
* Support search by various parameter combinations
* Enable efficient cache warming and invalidation

**Session and User Data**

* Cache user profiles with multiple lookup patterns
* Support search by user attributes
* Maintain session-specific and global data separation

**Configuration Best Practices**

**17. Performance Tuning**

**TTL Configuration**

CacheConfiguration.builder()  
 .localCacheTtl(300\_000L) // 5 minutes - frequent access  
 .remoteCacheTtl(3\_600\_000L) // 1 hour - shared data  
 .databaseCacheTtl(24 \* 3\_600\_000L) // 24 hours - persistent

**Size Management**

.maxLocalCacheSize(10\_000L) // Balance memory vs. hit rate  
.parameterCacheSize(20\_000L) // Larger for complex searches

**Strategy Selection**

* **High Redis Reliability**: Use REDIS\_THEN\_DATABASE
* **Database as Source of Truth**: Use DATABASE\_THEN\_REDIS
* **Performance Critical**: Use with manual fallback REDIS\_ONLY
* **Cost Sensitive**: Use with larger local caches DATABASE\_ONLY

The Transparent Hierarchical Cache represents a sophisticated solution for modern application caching needs, providing high performance through intelligent layering while supporting complex search and access patterns through its advanced parameter system.