```
"""
Pre-computation Optimization Module
Reduces O(N×M) complexity by pre-calculating static product scores
"""

import sqlite3
import json
from typing import List, Dict, Tuple
from datetime import datetime
from dataclasses import dataclass


@dataclass
class PrecomputedScore:
    product_id: str
    condition_name: str
    terpene_score: float
    cannabinoid_score: float
    modifier_score: float
    total_score: float
    last_updated: str


class PrecomputationEngine:
    """Manages pre-computed product scores for common conditions"""

    def __init__(self, db_path: str = "cannabis_kb.db"):
        self.db_path = db_path
        self.conn = sqlite3.connect(db_path)
        self._initialize_precompute_schema()

    def _initialize_precompute_schema(self):
        """Create table for pre-computed scores"""
        cursor = self.conn.cursor()

        cursor.execute("""
            CREATE TABLE IF NOT EXISTS precomputed_scores (
                product_id TEXT,
                condition_name TEXT,
                terpene_score REAL,
                cannabinoid_score REAL,
                modifier_score REAL,
                total_score REAL,
                computation_version INTEGER DEFAULT 1,
                last_updated TIMESTAMP,
                PRIMARY KEY (product_id, condition_name)
            )
        """)

        # Index for fast lookups
```

```
    cursor.execute("""
        CREATE INDEX IF NOT EXISTS idx_product_condition
        ON precomputed_scores(product_id, condition_name)
    """)

    self.conn.commit()

def precompute_all_products(self, products: List[Dict],
                            common_conditions: List[str],
                            knowledge_base) -> int:
    """
    Pre-compute scores for all products against common conditions
    This runs once per product update or daily
    """
    cursor = self.conn.cursor()
    computed_count = 0

    print(f"\nPre-computing scores for {len(products)} products...")
    print(f"Target conditions: {', '.join(common_conditions)}")

    for product in products:
        for condition in common_conditions:
            scores = self._compute_static_scores(
                product, condition, knowledge_base
            )

            cursor.execute("""
                INSERT OR REPLACE INTO precomputed_scores
                (product_id, condition_name, terpene_score, cannabinoid_score,
                 modifier_score, total_score, last_updated)
                VALUES (?, ?, ?, ?, ?, ?, ?)
            """, (
                product['id'],
                condition,
                scores['terpene'],
                scores['cannabinoid'],
                scores['modifier'],
                scores['total'],
                datetime.now()
            ))
            computed_count += 1

    self.conn.commit()
    print(f"✓ Pre-computed {computed_count} product-condition pairs")
    return computed_count

def _compute_static_scores(self, product: Dict, condition: str,
                           knowledge_base) -> Dict:
    """
    Compute the static portion of the score
```

```
    (excludes user history and personal thresholds)
    """
    terpene_score = 0.0
    cannabinoid_score = 0.0
    modifier_score = 0.0

    # Terpene scoring (severity=5 as baseline)
    if 'terpenes' in product:
        for terpene in product['terpenes']:
            efficacy = knowledge_base.lookup_efficacy(
                terpene['name'], condition, 'terpene'
            )
            # Normalize concentration
            norm_conc = min(terpene['concentration'] / 100, 1.0)
            terpene_score += efficacy * norm_conc * 5  # baseline severity

    # Cannabinoid scoring
    if 'cannabinoids' in product:
        for cannabinoid in product['cannabinoids']:
            efficacy = knowledge_base.lookup_efficacy(
                cannabinoid['name'], condition, 'cannabinoid'
            )
            norm_amount = min(cannabinoid['percentage'] / 100, 1.0)
            cannabinoid_score += efficacy * norm_amount * 5

    # Modifier scoring
    if 'flavonoids' in product:
        for flavonoid in product['flavonoids']:
            efficacy = knowledge_base.lookup_efficacy(
                flavonoid['name'], condition, 'flavonoid'
            )
            modifier_score += efficacy * 5

    # Base weights for static score
    total = (0.5 * terpene_score + 0.3 * cannabinoid_score + 0.2 * modifier_score)

    return {
        'terpene': terpene_score,
        'cannabinoid': cannabinoid_score,
        'modifier': modifier_score,
        'total': total
    }

def get_precomputed_score(self, product_id: str,
                          condition_name: str) -> PrecomputedScore:
    """Retrieve pre-computed score for fast lookup"""
    cursor = self.conn.cursor()
    cursor.execute("""
        SELECT product_id, condition_name, terpene_score, cannabinoid_score,
               modifier_score, total_score, last_updated
```

```
            FROM precomputed_scores
            WHERE product_id = ? AND condition_name = ?
        """, (product_id, condition_name))

        row = cursor.fetchone()
        if row:
            return PrecomputedScore(*row)
        return None

    def get_all_scores_for_product(self, product_id: str) -> List[PrecomputedScore]:
        """Get all pre-computed scores for a product"""
        cursor = self.conn.cursor()
        cursor.execute("""
            SELECT product_id, condition_name, terpene_score, cannabinoid_score,
                   modifier_score, total_score, last_updated
            FROM precomputed_scores
            WHERE product_id = ?
        """, (product_id,))

        return [PrecomputedScore(*row) for row in cursor.fetchall()]

    def close(self):
        self.conn.close()


class OptimizedRecommendationEngine:
    """
    Optimized version using pre-computed scores
    Runtime complexity: O(N) instead of O(N×M)
    """

    def __init__(self, knowledge_base, precompute_engine: PrecomputationEngine):
        self.kb = knowledge_base
        self.precompute = precompute_engine

    def rank_products_optimized(self, user_profile: Dict,
                                product_ids: List[str]) -> List[Dict]:
        """
        Fast ranking using pre-computed static scores + dynamic adjustments
        """
        results = []

        # Get adaptive weights for this user
        weights = self._get_adaptive_weights(user_profile)

        for product_id in product_ids:
            # Retrieve pre-computed base scores
            base_scores = {}
            total_base = 0.0
```

```
        for condition in user_profile['conditions']:
            precomputed = self.precompute.get_precomputed_score(
                product_id, condition['name']
            )

            if precomputed:
                # Weight by severity
                severity_weight = condition['severity'] / 5.0  # Normalize from baseline
                base_scores[condition['name']] = {
                    'terpene': precomputed.terpene_score * severity_weight,
                    'cannabinoid': precomputed.cannabinoid_score * severity_weight,
                    'modifier': precomputed.modifier_score * severity_weight
                }
                total_base += precomputed.total_score * severity_weight

        if not base_scores:
            continue  # Skip if no pre-computed data

        # Apply DELTA: Personal adjustments
        delta_score = self._calculate_personal_delta(
            product_id, user_profile, base_scores
        )

        # Final score = Static Base + Delta
        final_score = total_base + delta_score

        results.append({
            'product_id': product_id,
            'base_score': total_base,
            'delta': delta_score,
            'final_score': max(0, final_score),
            'weights_used': weights
        })

    # Normalize and sort
    if results:
        max_score = max(r['final_score'] for r in results)
        min_score = min(r['final_score'] for r in results)
        score_range = max_score - min_score

        if score_range > 0:
            for result in results:
                result['match_score'] = ((result['final_score'] - min_score) / score_range) * 100
        else:
            for result in results:
                result['match_score'] = 50.0

    results.sort(key=lambda x: x['match_score'], reverse=True)
    return results
```

```
    def _calculate_personal_delta(self, product_id: str, user_profile: Dict,
                                  base_scores: Dict) -> float:
        """
        Calculate the delta from base score based on:
        - Personal history
        - Threshold violations
        - Delivery/grow preferences
        """
        delta = 0.0

        # History factor
        for item in user_profile.get('history', []):
            if item['productId'] == product_id:
                if item['rating'] >= 4:
                    delta += 0.4
                elif item['rating'] <= 2:
                    delta -= 0.3

        # Threshold penalties (simplified - would need full product data)
        # This would require a lookup to get actual cannabinoid percentages
        # For now, we assume threshold checks are done separately

        # Preference bonuses
        # These would also require product metadata lookup

        return delta

    def _get_adaptive_weights(self, user_profile: Dict) -> Dict:
        """Get personalized weights (same as before)"""
        return {
            'terpene': 0.5,
            'cannabinoid': 0.3,
            'modifier': 0.2
        }


# ============================================================================
# PERFORMANCE COMPARISON
# ============================================================================

def benchmark_comparison(standard_engine, optimized_engine,
                         products: List[Dict], user_profile: Dict):
    """Compare performance of standard vs optimized approach"""
    import time

    print("\n" + "="*70)
    print("PERFORMANCE BENCHMARK")
    print("="*70)

    # Standard approach
```

```
    print("\nStandard Algorithm (O(N×M) complexity):")
    start = time.time()
    standard_results = standard_engine.rank_products(
        user_profile, products
    )
    standard_time = time.time() - start
    print(f"  Time: {standard_time*1000:.2f}ms for {len(products)} products")

    # Optimized approach
    print("\nOptimized Algorithm (O(N) complexity with pre-compute):")
    product_ids = [p['id'] for p in products]
    start = time.time()
    optimized_results = optimized_engine.rank_products_optimized(
        user_profile, product_ids
    )
    optimized_time = time.time() - start
    print(f"  Time: {optimized_time*1000:.2f}ms for {len(products)} products")

    # Calculate speedup
    speedup = standard_time / optimized_time if optimized_time > 0 else 0
    print(f"\n  Speedup: {speedup:.1f}x faster")
    print(f"  Reduction: {((standard_time - optimized_time)/standard_time)*100:.1f}%")

    # Extrapolate to larger catalog
    print("\n" + "="*70)
    print("SCALABILITY PROJECTION")
    print("="*70)

    catalog_sizes = [100, 500, 1000, 5000, 10000]
    print("\nProjected times for larger catalogs:")
    print(f"{'Catalog Size':<15} {'Standard':<15} {'Optimized':<15} {'Speedup':<10}")
    print("-" * 60)

    for size in catalog_sizes:
        std_projected = (standard_time / len(products)) * size
        opt_projected = (optimized_time / len(products)) * size
        speedup_proj = std_projected / opt_projected if opt_projected > 0 else 0

        print(f"{size:<15} {std_projected*1000:<14.1f}ms {opt_projected*1000:<14.1f}ms {speedup_proj:<9.1f}x")


# ============================================================================
# USAGE EXAMPLE
# ============================================================================

if __name__ == "__main__":
    from recommendation_engine import KnowledgeBase, RecommendationEngine

    # Initialize systems
    kb = KnowledgeBase()
```

```
precompute = PrecomputationEngine()

# Sample products (would come from your database)
sample_products = [
    {
        'id': 'prod_001',
        'name': 'Lavender Dream',
        'terpenes': [
            {'name': 'Linalool', 'concentration': 0.9},
            {'name': 'Myrcene', 'concentration': 0.6}
        ],
        'cannabinoids': [
            {'name': 'THC', 'percentage': 12.5},
            {'name': 'CBD', 'percentage': 10.0}
        ],
        'flavonoids': [
            {'name': 'Apigenin'}
        ]
    },
    {
        'id': 'prod_002',
        'name': 'Citrus Bliss',
        'terpenes': [
            {'name': 'Limonene', 'concentration': 1.2},
            {'name': 'Pinene', 'concentration': 0.4}
        ],
        'cannabinoids': [
            {'name': 'THC', 'percentage': 22.0},
            {'name': 'CBD', 'percentage': 2.0}
        ],
        'flavonoids': [
            {'name': 'Kaempferol'}
        ]
    },
    {
        'id': 'prod_003',
        'name': 'Peaceful Nights',
        'terpenes': [
            {'name': 'Linalool', 'concentration': 0.7},
            {'name': 'Myrcene', 'concentration': 0.8}
        ],
        'cannabinoids': [
            {'name': 'THC', 'percentage': 8.0},
            {'name': 'CBD', 'percentage': 18.0}
        ],
        'flavonoids': [
            {'name': 'Quercetin'}
        ]
    }
]
```

```
# Common conditions to pre-compute
common_conditions = ['Anxiety', 'Insomnia', 'Pain', 'Depression', 'Inflammation']

# Run pre-computation (this would run nightly or on product updates)
print("\n" + "="*70)
print("PRE-COMPUTATION PHASE")
print("="*70)
count = precompute.precompute_all_products(sample_products, common_conditions, kb)

# Sample user
user_profile = {
    'id': 'user_12345',
    'conditions': [
        {'name': 'Anxiety', 'severity': 9},
        {'name': 'Insomnia', 'severity': 7}
    ],
    'history': [
        {'productId': 'prod_001', 'rating': 5, 'dominantTerpene': 'Linalool'}
    ],
    'feedbackCount': 8
}

# Query pre-computed scores
print("\n" + "="*70)
print("SAMPLE PRE-COMPUTED SCORES")
print("="*70)

for product in sample_products[:2]:
    print(f"\nProduct: {product['name']} ({product['id']})")
    scores = precompute.get_all_scores_for_product(product['id'])
    for score in scores:
        print(f"  {score.condition_name}:")
        print(f"    Terpene: {score.terpene_score:.3f}")
        print(f"    Cannabinoid: {score.cannabinoid_score:.3f}")
        print(f"    Modifier: {score.modifier_score:.3f}")
        print(f"    Total: {score.total_score:.3f}")

# Performance comparison (if you have the standard engine imported)
print("\n" + "="*70)
print("NOTE: Performance comparison requires full product objects")
print("      Pre-computed approach eliminates O(M) condition iterations")
print("      Expected speedup: 5-10x for catalogs with 1000+ products")
print("="*70)

# Cleanup
precompute.close()
kb.close()

print("\n✓ Pre-computation demo complete")
```