Source: C:\Users\Admin\Desktop\CMC\ms1_emailIngestion\core\queue_manager.py

```
"""
Queue Manager
High-performance queue system v■i Redis backing
H■ tr■ priority queue và batch processing
"""
import json
import time
from typing import List, Dict, Optional
from datetime import datetime, timezone
from concurrent_storage.redis_manager import get_redis_storage


class EmailQueue:
    """
    Redis-backed queue cho email processing
    S■ d■ng Redis Sorted Set cho priority queue
    """

    QUEUE_KEY = "queue:emails"
    PROCESSING_KEY = "queue:processing"
    FAILED_KEY = "queue:failed"

    def __init__(self):
        self.redis = get_redis_storage()

    def enqueue(self, email_id: str, email_data: Dict, priority: Optional[float] = None) -> Optional
        """
        Thêm email vào queue

        Args:
            email_id: Email ID
            email_data: Full email data (subject, body, attachments, etc.)
            priority: Priority score (lower = higher priority)
                     None = timestamp (FIFO)

        Returns:
            email_id if enqueued, None if already in queue/processed
        """
        # Ki■m tra ■ã processed ch■a
        if self.redis.is_email_processed(email_id):
            return None

        # Ki■m tra ■ã trong queue ch■a
        if self.redis.redis.zscore(self.QUEUE_KEY, email_id) is not None:
            return None

        # Set priority
        if priority is None:
            priority = datetime.now(timezone.utc).timestamp()

        # Store email data
        data_key = f"email:data:{email_id}"
        self.redis.redis.setex(
            data_key,
            3600 * 24,  # TTL 24h
            json.dumps(email_data, default=str)
        )

        # Add to queue
        self.redis.redis.zadd(self.QUEUE_KEY, {email_id: priority})

        return email_id

    def enqueue_batch(self, emails: List[tuple]) -> List[str]:
        """
        Batch enqueue multiple emails

        Args:
            emails: List of (email_id, email_data, priority)

        Returns:
            List of email_ids that were successfully enqueued
        """
        enqueued_ids = []

        # Batch check processed
        email_ids = [e[0] for e in emails]
        processed_status = self._batch_check_processed(email_ids)

        # Batch check queue existence
        pipeline = self.redis.redis.pipeline()
        for email_id in email_ids:
            pipeline.zscore(self.QUEUE_KEY, email_id)
        in_queue = pipeline.execute()

        # Prepare batch insert
        to_insert = {}
        to_store = {}

        for i, (email_id, email_data, priority) in enumerate(emails):
            # Skip if processed or already in queue
            if processed_status[i] or in_queue[i] is not None:
                continue

            if priority is None:
                priority = datetime.now(timezone.utc).timestamp() + i * 0.001

            to_insert[email_id] = priority
            to_store[email_id] = email_data
            enqueued_ids.append(email_id)

        if not to_insert:
            return []

        # Batch insert
        pipeline = self.redis.redis.pipeline()

        # Store email data
        for email_id, email_data in to_store.items():
            data_key = f"email:data:{email_id}"
            pipeline.setex(
                data_key,
                3600 * 24,
                json.dumps(email_data, default=str)
            )

        # Add to queue
        pipeline.zadd(self.QUEUE_KEY, to_insert)
        pipeline.execute()

        return enqueued_ids

    def dequeue_batch(self, batch_size: int = 50) -> List[tuple]:
        """
        L■y batch emails t■ queue (oldest/highest priority first)

        Args:
            batch_size: Number of emails to dequeue

        Returns:
            List of (email_id, email_data)
        """
        # Get email IDs (atomic pop)
        email_ids = self._atomic_pop(batch_size)

        if not email_ids:
            return []

        # Batch fetch email data
        pipeline = self.redis.redis.pipeline()
        for email_id in email_ids:
            data_key = f"email:data:{email_id}"
            pipeline.get(data_key)

        email_data_list = pipeline.execute()

        # Parse and return
        result = []
        for email_id, email_data_json in zip(email_ids, email_data_list):
            if email_data_json:
                email_data = json.loads(email_data_json)
                result.append((email_id, email_data))
            else:
                # Data missing, re-queue
                self.redis.redis.zadd(
                    self.QUEUE_KEY,
                    {email_id: datetime.now(timezone.utc).timestamp()}
                )

        return result

    def _atomic_pop(self, count: int) -> List[str]:
        """
        Atomic pop from queue using Lua script
        Moves emails from queue to processing set
        """
        lua_script = """
        local queue_key = KEYS[1]
        local processing_key = KEYS[2]
        local count = tonumber(ARGV[1])
        local timestamp = tonumber(ARGV[2])

        -- Get emails from queue (lowest score = highest priority)
        local email_ids = redis.call('ZRANGE', queue_key, 0, count - 1)

        if #email_ids == 0 then
            return {}
        end

        -- Remove from queue
        redis.call('ZREM', queue_key, unpack(email_ids))

        -- Add to processing set with timeout timestamp
        local processing_items = {}
        for i, email_id in ipairs(email_ids) do
            table.insert(processing_items, timestamp + 300)  -- 5 min timeout
            table.insert(processing_items, email_id)
        end
        redis.call('ZADD', processing_key, unpack(processing_items))

        return email_ids
        """

        try:
            result = self.redis.redis.eval(
                lua_script,
                2,
                self.QUEUE_KEY,
                self.PROCESSING_KEY,
                count,
                datetime.now(timezone.utc).timestamp()
            )
            return result if result else []
        except Exception as e:
            print(f"[EmailQueue] Atomic pop error: {e}")
            return []

    def mark_processed(self, email_ids: List[str]):
        """
        Mark emails as processed (batch)
        Remove from processing set
        """
        if not email_ids:
            return

        pipeline = self.redis.redis.pipeline()

        # Remove from processing
        pipeline.zrem(self.PROCESSING_KEY, *email_ids)

        # Clean up data
        for email_id in email_ids:
            data_key = f"email:data:{email_id}"
            pipeline.delete(data_key)

        pipeline.execute()
    def mark_failed(self, email_id: str, error: str):
        """
        Mark email as failed
        Move to failed queue for retry/investigation
        """
        # Remove from processing
        self.redis.redis.zrem(self.PROCESSING_KEY, email_id)

        # Add to failed with metadata
        failed_data = {
            "email_id": email_id,
            "error": error,
            "timestamp": datetime.now(timezone.utc).isoformat(),
            "retry_count": 0
        }

        self.redis.redis.zadd(
            self.FAILED_KEY,
            {json.dumps(failed_data): datetime.now(timezone.utc).timestamp()}
        )

    def requeue_timeouts(self):
        """
        Re-queue emails that timed out in processing
        Run periodically (e.g., every 1 minute)
        """
        now = datetime.now(timezone.utc).timestamp()

        # Find timed out emails
        timed_out = self.redis.redis.zrangebyscore(
            self.PROCESSING_KEY,
            0,
            now
        )

        if not timed_out:
            return 0

        # Move back to queue
        pipeline = self.redis.redis.pipeline()

        for email_id in timed_out:
            # Remove from processing
            pipeline.zrem(self.PROCESSING_KEY, email_id)
            # Add back to queue with current timestamp
            pipeline.zadd(self.QUEUE_KEY, {email_id: now})

        pipeline.execute()

        print(f"[EmailQueue] Re-queued {len(timed_out)} timed out emails")
        return len(timed_out)

    def is_in_queue(self, email_id: str) -> bool:
        """
        Ki■m tra email ■ã có trong hàng ■■i (queue) ho■c ■ang ■■■c x■ lý (processing) ch■a.

        Args:
            email_id: ID c■a email c■n ki■m tra.

        Returns:
            True n■u email t■n t■i trong queue ho■c processing, ng■■c l■i là False.
        """
        # Check in the main queue (Sorted Set)
        if self.redis.redis.zscore(self.QUEUE_KEY, email_id) is not None:
            return True

        # Check in the processing queue (Sorted Set)
        return self.redis.redis.zscore(self.PROCESSING_KEY, email_id) is not None

    def get_stats(self) -> Dict:
        """Get queue statistics"""
        return {
            "queue_size": self.redis.redis.zcard(self.QUEUE_KEY),
            "processing_size": self.redis.redis.zcard(self.PROCESSING_KEY),
            "failed_size": self.redis.redis.zcard(self.FAILED_KEY),
            "timestamp": datetime.now(timezone.utc).isoformat()
        }
    def _batch_check_processed(self, email_ids: List[str]) -> List[bool]:
        """Batch check if emails are processed"""
        if not email_ids:
            return []

        # Use SMISMEMBER (Redis 6.2+) or fallback to pipeline
        try:
            # Redis 6.2+ supports SMISMEMBER
            results = self.redis.redis.smismember(
                self.redis.KEY_PROCESSED,
                email_ids
            )
            return [bool(r) for r in results]
        except:
            # Fallback for older Redis
            pipeline = self.redis.redis.pipeline()
            for email_id in email_ids:
                pipeline.sismember(self.redis.KEY_PROCESSED, email_id)
            return [bool(r) for r in pipeline.execute()]


# Singleton
_queue_instance = None

def get_email_queue() -> EmailQueue:
    """Get singleton EmailQueue instance"""
    global _queue_instance
    if _queue_instance is None:
        _queue_instance = EmailQueue()
    return _queue_instance
```