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

```
"""
Batch Email Processor
X■ lý email song song v■i ThreadPoolExecutor
T■i ■u hóa I/O operations
"""
import time
import threading
from concurrent.futures import ThreadPoolExecutor, as_completed
from typing import List, Dict, Optional
from datetime import datetime, timezone

from core.queue_manager import get_email_queue
from core.unified_email_processor import EmailProcessor
from core.session_manager import session_manager
from core.token_manager import get_token


class BatchEmailProcessor:
    """
    High-performance batch processor
    X■ lý N emails song song v■i ThreadPool
    """

    def __init__(
        self,
        batch_size: int = 50,
        max_workers: int = 20,
        fetch_interval: float = 2.0
    ):
        """
        Args:
            batch_size: Number of emails per batch
            max_workers: Number of parallel workers
            fetch_interval: Seconds between queue checks
        """
        self.batch_size = batch_size
        self.max_workers = max_workers
        self.fetch_interval = fetch_interval

        self.queue = get_email_queue()
        self.processor: Optional[EmailProcessor] = None
        self.executor: Optional[ThreadPoolExecutor] = None

        self.active = False
        self.thread: Optional[threading.Thread] = None
        self._stop_event = threading.Event()

        # Stats
        self.stats = {
            "batches_processed": 0,
            "emails_success": 0,
            "emails_failed": 0,
            "total_processing_time": 0.0,
            "avg_batch_time": 0.0
        }

    def start(self) -> bool:
        """Start batch processor"""
        if self.active:
            print("[BatchProcessor] Already active")
            return False

        print(f"[BatchProcessor] Starting...")
        print(f"  Batch size: {self.batch_size}")
        print(f"  Workers: {self.max_workers}")
        print(f"  Fetch interval: {self.fetch_interval}s")

        # Initialize processor
        token = get_token()
        self.processor = EmailProcessor(token)

        # Initialize executor
        self.executor = ThreadPoolExecutor(
            max_workers=self.max_workers,
            thread_name_prefix="EmailWorker"
        )

        # Start processing loop
        self.active = True
        self._stop_event.clear()
        self.thread = threading.Thread(
            target=self._processing_loop,
            daemon=True,
            name="BatchProcessorLoop"
        )
        self.thread.start()

        print("[BatchProcessor] Started successfully")
        return True

    def stop(self):
        """Stop batch processor"""
        if not self.active:
            return

        print("[BatchProcessor] Stopping...")
        self.active = False
        self._stop_event.set()

        # Wait for thread
        if self.thread and self.thread.is_alive():
            self.thread.join(timeout=10)

        # Shutdown executor
        if self.executor:
            self.executor.shutdown(wait=True, cancel_futures=False)
            self.executor = None

        print("[BatchProcessor] Stopped")
        self._print_stats()

    def _processing_loop(self):
        """Main processing loop"""
        print("[BatchProcessor] Processing loop started")

        while self.active and not self._stop_event.is_set():
            try:
                # Check queue stats
                queue_stats = self.queue.get_stats()
                queue_size = queue_stats["queue_size"]
                shutting_down = self._stop_event.is_set()

                # Ch■ x■ lý khi có ■■ 1 batch, ho■c khi ■ang shutdown và có email t■n
                if queue_size < self.batch_size and not (shutting_down and queue_size > 0):
                    # Ch■a ■■ batch, ■■i
                    time.sleep(self.fetch_interval)
                    continue

                if queue_size > 0:
                    print(f"\n[BatchProcessor] Queue size: {queue_size}. Triggering batch processing

                # Fetch batch
                batch = self.queue.dequeue_batch(self.batch_size)

                if not batch:
                    time.sleep(self.fetch_interval)
                    continue

                print(f"[BatchProcessor] Processing batch of {len(batch)} emails...")

                # Process batch
                batch_start = time.time()
                result = self._process_batch_parallel(batch)
                batch_time = time.time() - batch_start

                # Update stats
                self.stats["batches_processed"] += 1
                self.stats["emails_success"] += result["success"]
                self.stats["emails_failed"] += result["failed"]
                self.stats["total_processing_time"] += batch_time
                self.stats["avg_batch_time"] = (
                    self.stats["total_processing_time"] /
                    self.stats["batches_processed"]
                )

                print(f"[BatchProcessor] Batch completed in {batch_time:.2f}s")
                print(f"  Success: {result['success']}")
                print(f"  Failed: {result['failed']}")
                print(f"  Rate: {len(batch)/batch_time:.1f} emails/s")

                # Re-queue timeouts periodically
                if self.stats["batches_processed"] % 10 == 0:
                    self.queue.requeue_timeouts()

                # Brief pause before next batch
                time.sleep(0.5)

            except Exception as e:
                print(f"[BatchProcessor] Loop error: {e}")
                time.sleep(5)

        print("[BatchProcessor] Processing loop stopped")

    def _process_batch_parallel(self, batch: List[tuple]) -> Dict:
        """
        Process batch of emails in parallel

        Args:
            batch: List of (email_id, email_data)

        Returns:
            {"success": int, "failed": int}
        """
        result = {"success": 0, "failed": 0}

        # Submit all tasks
        futures = {}
        for email_id, email_data in batch:
            future = self.executor.submit(
                self._process_single_email,
                email_id,
                email_data
            )
            futures[future] = email_id

        # Collect results
        processed_ids = []

        for future in as_completed(futures):
            email_id = futures[future]
            try:
                success = future.result(timeout=30)

                if success:
                    result["success"] += 1
                    processed_ids.append(email_id)
                else:
                    result["failed"] += 1
                    self.queue.mark_failed(email_id, "Processing failed")

            except Exception as e:
                result["failed"] += 1
                self.queue.mark_failed(email_id, str(e))
                print(f"[BatchProcessor] Error processing {email_id}: {e}")

        # Batch mark processed
        if processed_ids:
            self.queue.mark_processed(processed_ids)

        return result

    def _process_single_email(self, email_id: str, email_data: Dict) -> bool:
        """
        Process single email (runs in thread pool)

        Args:
            email_id: Email ID
            email_data: Email data dict

        Returns:
            True if success
        """
        try:
            # Use the unified processor
            success = self.processor.process_email(
                message=email_data,
                source="batch_processor"
            )

            return success

        except Exception as e:
            print(f"[BatchProcessor] Email {email_id} error: {e}")
            return False

    def get_stats(self) -> Dict:
        """Get processor statistics"""
        return {
            "active": self.active,
            "batch_size": self.batch_size,
            "max_workers": self.max_workers,
            **self.stats,
            "queue_stats": self.queue.get_stats()
        }

    def _print_stats(self):
        """Print final statistics"""
        print("\n" + "=" * 70)
        print("BATCH PROCESSOR STATISTICS")
        print("=" * 70)
        print(f"Batches Processed: {self.stats['batches_processed']}")
        print(f"Emails Success: {self.stats['emails_success']}")
        print(f"Emails Failed: {self.stats['emails_failed']}")
        print(f"Avg Batch Time: {self.stats['avg_batch_time']:.2f}s")

        if self.stats['total_processing_time'] > 0:
            total_emails = self.stats['emails_success'] + self.stats['emails_failed']
            throughput = total_emails / self.stats['total_processing_time']
            print(f"Throughput: {throughput:.1f} emails/s")

        print("=" * 70)


# Singleton
_batch_processor_instance = None

def get_batch_processor(
    batch_size: int = 50,
    max_workers: int = 20
) -> BatchEmailProcessor:
    """Get singleton BatchEmailProcessor"""
    global _batch_processor_instance
    if _batch_processor_instance is None:
        _batch_processor_instance = BatchEmailProcessor(
            batch_size=batch_size,
            max_workers=max_workers
        )
    return _batch_processor_instance
```