Source: C:\Users\Admin\Desktop\CMC\ms1_emailIngestion\concurrent_storage\redis_manager.py

```
"""
redis_storage_manager.py
Redis-based storage thay th■ JSON files
C■i thi■n performance, concurrency và scalability
"""
import json
import redis
from typing import Optional, Set, Dict, List, Any
from datetime import datetime, timezone, timedelta
from contextlib import contextmanager


class RedisStorageManager:
    """
    Centralized Redis storage manager
    Thay th■ t■t c■ JSON files b■ng Redis structures
    """
    # Redis key prefixes
    KEY_PROCESSED = "email:processed"
    KEY_PENDING = "email:pending"
    KEY_FAILED = "email:failed"
    KEY_SESSION_CURRENT = "session:current"
    KEY_SESSION_PREFIX = "session:"
    KEY_SESSIONS_HISTORY = "sessions:history"
    KEY_WEBHOOK_SUB = "webhook:subscription"
    KEY_REFRESH_TOKEN = "auth:refresh_token"
    KEY_LOCK_PREFIX = "lock:"
    KEY_METRICS_PREFIX = "metrics:"
    KEY_COUNTER_PREFIX = "counter:"
    KEY_RATELIMIT_PREFIX = "ratelimit:"

    # TTL defaults (seconds)
    TTL_PROCESSED_EMAILS = 30 * 24 * 3600  # 30 days
    TTL_SESSION = 7 * 24 * 3600  # 7 days
    TTL_LOCK = 30  # 30 seconds
    TTL_RATELIMIT = 3600  # 1 hour

    def __init__(
        self,
        host: str = "localhost",
        port: int = 6379,
        db: int = 0,
        password: Optional[str] = None,
        decode_responses: bool = True
    ):
        """
        Initialize Redis connection

        Args:
            host: Redis host
            port: Redis port
            db: Redis database number
            password: Redis password (if any)
            decode_responses: Auto-decode bytes to str
        """
        self.redis = redis.Redis(
            host=host,
            port=port,
            db=db,
            password=password,
            decode_responses=decode_responses,
            socket_connect_timeout=5,
            socket_keepalive=True,
            health_check_interval=30
        )

        # Test connection
        try:
            self.redis.ping()
            print("[RedisStorage] Connected successfully")
        except redis.ConnectionError as e:
            raise Exception(f"Failed to connect to Redis: {e}")
```

```
# ==================== PROCESSED EMAIL IDs ====================

def is_email_processed(self, email_id: str) -> bool:
    """
    Ki■m tra email ■ã ■■■c x■ lý ch■a
    O(1) operation v■i Redis SET
    """
    return self.redis.sismember(self.KEY_PROCESSED, email_id)

def mark_email_processed(self, email_id: str, ttl: Optional[int] = None) -> bool:
    """
    ■ánh d■u email ■ã x■ lý

    Args:
        email_id: Email ID
        ttl: Custom TTL (seconds), m■c ■■nh 30 ngày

    Returns:
        True n■u email ch■a processed tr■■c ■ó
    """
    # SADD returns 1 if new, 0 if already exists
    result = self.redis.sadd(self.KEY_PROCESSED, email_id)

    # Set TTL cho key (ch■ c■n set 1 l■n)
    if ttl is None:
        ttl = self.TTL_PROCESSED_EMAILS
    self.redis.expire(self.KEY_PROCESSED, ttl)

    return result == 1

def get_processed_count(self) -> int:
    """S■ l■■ng emails ■ã x■ lý"""
    return self.redis.scard(self.KEY_PROCESSED)

def get_processed_emails(self, limit: int = 100) -> Set[str]:
    """
    L■y danh sách emails ■ã x■ lý (for debugging)

    Args:
        limit: Gi■i h■n s■ l■■ng
    """
    return set(self.redis.srandmember(self.KEY_PROCESSED, limit))

def cleanup_old_processed(self, days: int = 30):
    """
    Cleanup emails processed quá X ngày
    (Th■c t■ không c■n vì ■ã có TTL t■ ■■ng)
    """
    # Redis SET v■i TTL ■ã t■ ■■ng cleanup
    # Method này ch■ ■■ manual cleanup n■u c■n
    pass

# ==================== PENDING QUEUE ====================

def add_pending_email(self, email_id: str, priority: Optional[float] = None):
    """
    Thêm email vào pending queue

    Args:
        email_id: Email ID
        priority: Priority score (timestamp), lower = higher priority
    """
    if priority is None:
        priority = datetime.now(timezone.utc).timestamp()

    # Dùng Sorted Set ■■ có th■ sort by priority
    self.redis.zadd(self.KEY_PENDING, {email_id: priority})

def get_next_pending(self, count: int = 1) -> List[str]:
    """
    L■y email pending ti■p theo (oldest first)

    Args:
        count: S■ l■■ng emails c■n l■y

    Returns:
        List email IDs
```

```
    """
    # ZRANGE v■i WITHSCORES=False ■■ ch■ l■y IDs
    return self.redis.zrange(self.KEY_PENDING, 0, count - 1)

def remove_pending(self, email_id: str) -> bool:
    """Xóa email kh■i pending queue"""
    return self.redis.zrem(self.KEY_PENDING, email_id) > 0

def get_pending_count(self) -> int:
    """S■ l■■ng emails pending"""
    return self.redis.zcard(self.KEY_PENDING)

def move_to_failed(self, email_id: str, error: str):
    """
    Di chuy■n email sang failed queue (DLQ)

    Args:
        email_id: Email ID
        error: Error message
    """
    # Remove from pending
    self.redis.zrem(self.KEY_PENDING, email_id)

    # Add to failed list
    self.redis.lpush(self.KEY_FAILED, email_id)

    # Store error info
    retry_key = f"email:retry:{email_id}"
    self.redis.hincrby(retry_key, "count", 1)
    self.redis.hset(retry_key, "last_error", error)
    self.redis.hset(retry_key, "last_attempt", datetime.now(timezone.utc).isoformat())
    self.redis.expire(retry_key, 7 * 24 * 3600)  # 7 days

def get_failed_count(self) -> int:
    """S■ l■■ng failed emails"""
    return self.redis.llen(self.KEY_FAILED)

# ==================== SESSION STATE ====================

def set_session_state(self, session_data: Dict[str, Any]) -> bool:
    """
    L■u session state

    Args:
        session_data: Dict ch■a session info
    """
    # Convert all values to string for Redis Hash
    redis_data = {k: str(v) for k, v in session_data.items()}

    # HSET multiple fields
    self.redis.hset(self.KEY_SESSION_CURRENT, mapping=redis_data)

    # Set TTL
    self.redis.expire(self.KEY_SESSION_CURRENT, self.TTL_SESSION)

    return True

def get_session_state(self) -> Dict[str, Any]:
    """L■y session state hi■n t■i"""
    data = self.redis.hgetall(self.KEY_SESSION_CURRENT)
    if not data:
        return {}

    # Convert back to appropriate types
    return data

def update_session_field(self, field: str, value: Any) -> bool:
    """
    Update 1 field c■a session (partial update)

    Args:
        field: Field name
        value: New value
    """
    self.redis.hset(self.KEY_SESSION_CURRENT, field, str(value))
    return True
def increment_session_counter(self, field: str, amount: int = 1) -> int:
```

```
    """
    Atomic increment counter trong session

    Args:
        field: Counter field name (e.g., "polling_errors")
        amount: Amount to increment

    Returns:
        New value after increment
    """
    return self.redis.hincrby(self.KEY_SESSION_CURRENT, field, amount)

def delete_session(self):
    """Xóa session hi■n t■i"""
    self.redis.delete(self.KEY_SESSION_CURRENT)

# ==================== SESSION HISTORY ====================

def save_session_history(self, session_data: Dict[str, Any], max_history: int = 100):
    """
    L■u session vào history

    Args:
        session_data: Session data
        max_history: Maximum number of sessions to keep
    """
    # Serialize to JSON
    session_json = json.dumps(session_data, default=str)

    # Add to history list (head)
    self.redis.lpush(self.KEY_SESSIONS_HISTORY, session_json)

    # Trim to keep only max_history items
    self.redis.ltrim(self.KEY_SESSIONS_HISTORY, 0, max_history - 1)

    # Also add to sorted set by timestamp for querying
    session_id = session_data.get("session_id")
    timestamp = session_data.get("start_time", datetime.now(timezone.utc).isoformat())
    if session_id:
        # Convert ISO timestamp to Unix timestamp
        if isinstance(timestamp, str):
            dt = datetime.fromisoformat(timestamp.replace("Z", "+00:00"))
            timestamp = dt.timestamp()
        self.redis.zadd("sessions:by_time", {session_id: timestamp})

def get_session_history(self, limit: int = 10) -> List[Dict]:
    """
    L■y session history

    Args:
        limit: Number of sessions to retrieve
    """
    sessions_json = self.redis.lrange(self.KEY_SESSIONS_HISTORY, 0, limit - 1)
    return [json.loads(s) for s in sessions_json]

# ==================== WEBHOOK SUBSCRIPTION ====================

def save_subscription(self, subscription_data: Dict[str, Any]):
    """L■u webhook subscription info"""
    redis_data = {k: str(v) for k, v in subscription_data.items()}
    self.redis.hset(self.KEY_WEBHOOK_SUB, mapping=redis_data)

    # Set TTL based on expiration
    expiration = subscription_data.get("expirationDateTime")
    if expiration:
        if isinstance(expiration, str):
            exp_dt = datetime.fromisoformat(expiration.replace("Z", "+00:00"))
            ttl = int((exp_dt - datetime.now(timezone.utc)).total_seconds())
            if ttl > 0:
                self.redis.expire(self.KEY_WEBHOOK_SUB, ttl)

def get_subscription(self) -> Dict[str, Any]:
    """L■y webhook subscription info"""
    return self.redis.hgetall(self.KEY_WEBHOOK_SUB)

def delete_subscription(self):
    """Xóa subscription"""
```

```
    self.redis.delete(self.KEY_WEBHOOK_SUB)

# ==================== DISTRIBUTED LOCKS ====================

@contextmanager
def acquire_lock(self, lock_name: str, ttl: int = 30, blocking: bool = True, timeout: int = 10):
    """
    Context manager cho distributed lock

    Args:
        lock_name: Lock name (e.g., "polling", "webhook")
        ttl: Lock TTL in seconds
        blocking: Block until lock acquired
        timeout: Max time to wait (if blocking)

    Usage:
        with redis_storage.acquire_lock("polling"):
            # Critical section
            fetch_and_process_emails()
    """
    lock_key = f"{self.KEY_LOCK_PREFIX}{lock_name}"
    lock_value = f"{datetime.now(timezone.utc).timestamp()}"

    # Try to acquire lock
    if blocking:
        start_time = datetime.now(timezone.utc)
        while True:
            acquired = self.redis.set(lock_key, lock_value, nx=True, ex=ttl)
            if acquired:
                break

            # Check timeout
            elapsed = (datetime.now(timezone.utc) - start_time).total_seconds()
            if elapsed >= timeout:
                raise TimeoutError(f"Failed to acquire lock '{lock_name}' after {timeout}s")

            # Wait before retry
            import time
            time.sleep(0.1)
    else:
        acquired = self.redis.set(lock_key, lock_value, nx=True, ex=ttl)
        if not acquired:
            raise RuntimeError(f"Lock '{lock_name}' already held")

    try:
        yield
    finally:
        # Release lock (only if we still own it)
        stored_value = self.redis.get(lock_key)
        if stored_value == lock_value:
            self.redis.delete(lock_key)

def is_locked(self, lock_name: str) -> bool:
    """Check if lock is currently held"""
    lock_key = f"{self.KEY_LOCK_PREFIX}{lock_name}"
    return self.redis.exists(lock_key) > 0

# ==================== RATE LIMITING ====================

def check_rate_limit(
    self,
    key: str,
    limit: int,
    window: int = 3600
) -> tuple[bool, int]:
    """
    Check rate limit

    Args:
        key: Rate limit key (e.g., "polling", "webhook")
        limit: Max requests per window
        window: Time window in seconds

    Returns:
        (allowed, current_count)
    """
    rate_key = f"{self.KEY_RATELIMIT_PREFIX}{key}"
```

```
    # Increment counter
    count = self.redis.incr(rate_key)

    # Set expiry on first request
    if count == 1:
        self.redis.expire(rate_key, window)

    # Check limit
    allowed = count <= limit
    return (allowed, count)

def reset_rate_limit(self, key: str):
    """Reset rate limit counter"""
    rate_key = f"{self.KEY_RATELIMIT_PREFIX}{key}"
    self.redis.delete(rate_key)

# ==================== METRICS ====================

def increment_metric(self, metric_name: str, date: Optional[str] = None, amount: int = 1):
    """
    Increment metric counter

    Args:
        metric_name: Metric name (e.g., "emails_processed", "emails_failed")
        date: Date string (YYYYMMDD), defaults to today
        amount: Amount to increment
    """
    if date is None:
        date = datetime.now(timezone.utc).strftime("%Y%m%d")

    metrics_key = f"{self.KEY_METRICS_PREFIX}{date}"
    self.redis.hincrby(metrics_key, metric_name, amount)

    # Set TTL to 90 days
    self.redis.expire(metrics_key, 90 * 24 * 3600)

def get_metrics(self, date: Optional[str] = None) -> Dict[str, int]:
    """
    Get metrics for a specific date

    Args:
        date: Date string (YYYYMMDD), defaults to today
    """
    if date is None:
        date = datetime.now(timezone.utc).strftime("%Y%m%d")

    metrics_key = f"{self.KEY_METRICS_PREFIX}{date}"
    metrics = self.redis.hgetall(metrics_key)

    # Convert to int
    return {k: int(v) for k, v in metrics.items()}

def increment_counter(self, counter_name: str) -> int:
    """
    Increment global counter (không có TTL)

    Args:
        counter_name: Counter name (e.g., "total_processed")

    Returns:
        New counter value
    """
    counter_key = f"{self.KEY_COUNTER_PREFIX}{counter_name}"
    return self.redis.incr(counter_key)

def get_counter(self, counter_name: str) -> int:
    """Get counter value"""
    counter_key = f"{self.KEY_COUNTER_PREFIX}{counter_name}"
    value = self.redis.get(counter_key)
    return int(value) if value else 0

def reset_counter(self, counter_name: str):
    """Reset counter to 0"""
    counter_key = f"{self.KEY_COUNTER_PREFIX}{counter_name}"
    self.redis.delete(counter_key)

# ==================== UTILITY METHODS ====================
```

```
    def health_check(self) -> Dict[str, Any]:
        """
        Check Redis health and return stats
        """
        try:
            info = self.redis.info()
            return {
                "status": "healthy",
                "redis_version": info.get("redis_version"),
                "used_memory_human": info.get("used_memory_human"),
                "connected_clients": info.get("connected_clients"),
                "uptime_days": info.get("uptime_in_days"),
                "processed_emails": self.get_processed_count(),
                "pending_emails": self.get_pending_count(),
                "failed_emails": self.get_failed_count()
            }
        except Exception as e:
            return {
                "status": "unhealthy",
                "error": str(e)
            }

    def flush_all(self, confirm: bool = False):
        """
        WARNING: Xóa toàn b■ data trong Redis
        Ch■ dùng cho testing
        """
        if not confirm:
            raise ValueError("Must pass confirm=True to flush all data")
        self.redis.flushdb()
        print("[RedisStorage] All data flushed")

    def get_all_keys(self, pattern: str = "*") -> List[str]:
        """
        Get all keys matching pattern
        WARNING: Expensive operation, ch■ dùng cho debugging
        """
        return self.redis.keys(pattern)

    def close(self):
        """Close Redis connection"""
        self.redis.close()


# ==================== FACTORY FUNCTION ====================

_redis_storage_instance = None

def get_redis_storage(
    host: str = None,
    port: int = None,
    password: str = None
) -> RedisStorageManager:
    """
    Get singleton Redis storage instance

    Usage:
        from concurrent_storage.redis_manager import get_redis_storage
        redis = get_redis_storage()
        redis.mark_email_processed(email_id)
    """
    global _redis_storage_instance

    if _redis_storage_instance is None:
        # Load from env if not provided
        import os
        from dotenv import load_dotenv
        load_dotenv()

        host = host or os.getenv("REDIS_HOST", "localhost")
        port = port or int(os.getenv("REDIS_PORT", "6379"))
        password = password or os.getenv("REDIS_PASSWORD")
        db = int(os.getenv("REDIS_DB", "0"))

        _redis_storage_instance = RedisStorageManager(
            host=host,
            port=port,
```

```
            db=db,
            password=password
        )

    return _redis_storage_instance
```