BANKING SYSTEM PROJECT - DAY 7: DATA STRUCTURES IMPLEMENTATION

==============================================================

OVERVIEW

--------

Day 7 focuses on implementing advanced data structures for transaction management, specifically Stack and Queue data structures. These structures provide efficient transaction reversal capabilities (Undo/Redo) and batch processing for settlement operations.

IMPLEMENTED DATA STRUCTURES

---------------------------

1. UNDO/REDO STACK IMPLEMENTATION

- UndoRedoStack.java: Stack-based transaction reversal system

- Thread-safe implementation using ConcurrentHashMap

- Account-specific transaction history management

- Atomic operations for transaction reversal

2. SETTLEMENT QUEUE IMPLEMENTATION

- SettlementQueue.java: Queue-based batch processing system

- BlockingQueue for thread-safe operations

- Concurrent transaction processing

- Batch settlement capabilities

UNDO/REDO STACK DETAILED IMPLEMENTATION

---------------------------------------

1. CLASS STRUCTURE

@Component

public class UndoRedoStack {

private final Map<String, Stack<Transaction>> undoStacks;

private final Map<String, Stack<Transaction>> redoStacks;

public UndoRedoStack() {

this.undoStacks = new ConcurrentHashMap<>();

this.redoStacks = new ConcurrentHashMap<>();

}

}

2. CORE STACK OPERATIONS

PUSH UNDO OPERATION:

public void pushUndo(String accountId, Transaction transaction) {

undoStacks.computeIfAbsent(accountId, k -> new Stack<>()).push(transaction);

// Clear redo stack when new transaction is added

redoStacks.remove(accountId);

}

POP UNDO OPERATION:

public Transaction popUndo(String accountId) {

Stack<Transaction> undoStack = undoStacks.get(accountId);

if (undoStack != null && !undoStack.isEmpty()) {

Transaction transaction = undoStack.pop();

// Move to redo stack

redoStacks.computeIfAbsent(accountId, k -> new Stack<>()).push(transaction);

return transaction;

}

return null;

}

POP REDO OPERATION:

public Transaction popRedo(String accountId) {

Stack<Transaction> redoStack = redoStacks.get(accountId);

if (redoStack != null && !redoStack.isEmpty()) {

Transaction transaction = redoStack.pop();

// Move back to undo stack

undoStacks.computeIfAbsent(accountId, k -> new Stack<>()).push(transaction);

return transaction;

}

return null;

}

3. STACK VALIDATION METHODS

CAN UNDO CHECK:

public boolean canUndo(String accountId) {

Stack<Transaction> undoStack = undoStacks.get(accountId);

return undoStack != null && !undoStack.isEmpty();

}

CAN REDO CHECK:

public boolean canRedo(String accountId) {

Stack<Transaction> redoStack = redoStacks.get(accountId);

return redoStack != null && !redoStack.isEmpty();

}

4. STACK SIZE MONITORING

GET UNDO STACK SIZE:

public int getUndoStackSize(String accountId) {

Stack<Transaction> undoStack = undoStacks.get(accountId);

return undoStack != null ? undoStack.size() : 0;

}

GET REDO STACK SIZE:

public int getRedoStackSize(String accountId) {

Stack<Transaction> redoStack = redoStacks.get(accountId);

return redoStack != null ? redoStack.size() : 0;

}

5. STACK MANAGEMENT

CLEAR STACKS FOR ACCOUNT:

public void clearStacks(String accountId) {

undoStacks.remove(accountId);

redoStacks.remove(accountId);

}

CLEAR ALL STACKS:

public void clearAllStacks() {

undoStacks.clear();

redoStacks.clear();

}

6. PEEK OPERATIONS

PEEK UNDO:

public Transaction peekUndo(String accountId) {

Stack<Transaction> undoStack = undoStacks.get(accountId);

return (undoStack != null && !undoStack.isEmpty()) ? undoStack.peek() : null;

}

PEEK REDO:

public Transaction peekRedo(String accountId) {

Stack<Transaction> redoStack = redoStacks.get(accountId);

return (redoStack != null && !redoStack.isEmpty()) ? redoStack.peek() : null;

}

SETTLEMENT QUEUE DETAILED IMPLEMENTATION

----------------------------------------

1. CLASS STRUCTURE

@Component

public class SettlementQueue {

private final BlockingQueue<Transaction> settlementQueue;

private final Map<String, Transaction> pendingTransactions;

public SettlementQueue() {

this.settlementQueue = new LinkedBlockingQueue<>();

this.pendingTransactions = new ConcurrentHashMap<>();

}

}

2. CORE QUEUE OPERATIONS

ENQUEUE OPERATION:

public void enqueue(Transaction transaction) {

try {

settlementQueue.put(transaction);

pendingTransactions.put(transaction.getTransactionId(), transaction);

} catch (InterruptedException e) {

Thread.currentThread().interrupt();

throw new RuntimeException("Failed to enqueue transaction", e);

}

}

DEQUEUE OPERATION:

public Transaction dequeue() {

try {

Transaction transaction = settlementQueue.take();

pendingTransactions.remove(transaction.getTransactionId());

return transaction;

} catch (InterruptedException e) {

Thread.currentThread().interrupt();

return null;

}

}

3. QUEUE MONITORING METHODS

PEEK OPERATION:

public Transaction peek() {

return settlementQueue.peek();

}

SIZE CHECK:

public int size() {

return settlementQueue.size();

}

IS EMPTY CHECK:

public boolean isEmpty() {

return settlementQueue.isEmpty();

}

4. PENDING TRANSACTION MANAGEMENT

GET PENDING TRANSACTIONS:

public List<Transaction> getPendingTransactions() {

return new ArrayList<>(pendingTransactions.values());

}

IS PENDING CHECK:

public boolean isPending(String transactionId) {

return pendingTransactions.containsKey(transactionId);

}

MARK AS PROCESSED:

public void markAsProcessed(String transactionId) {

pendingTransactions.remove(transactionId);

}

5. QUEUE MANAGEMENT

CLEAR QUEUE:

public void clear() {

settlementQueue.clear();

pendingTransactions.clear();

}

GET PENDING COUNT:

public int getPendingCount() {

return pendingTransactions.size();

}

6. BATCH OPERATIONS

DRAIN ALL:

public List<Transaction> drainAll() {

List<Transaction> transactions = new ArrayList<>();

settlementQueue.drainTo(transactions);

pendingTransactions.clear();

return transactions;

}

DRAIN TO LIMIT:

public List<Transaction> drainTo(int maxElements) {

List<Transaction> transactions = new ArrayList<>();

settlementQueue.drainTo(transactions, maxElements);

// Remove drained transactions from pending map

transactions.forEach(tx -> pendingTransactions.remove(tx.getTransactionId()));

return transactions;

}

TRANSACTION REVERSAL IMPLEMENTATION

-----------------------------------

1. UNDO TRANSACTION SERVICE

@Service

public class UndoTransactionService {

@Autowired

private UndoRedoStack undoRedoStack;

@Autowired

private AccountRepository accountRepository;

@Autowired

private TransactionRepository transactionRepository;

@Autowired

private AuditService auditService;

public Transaction undoLastTransaction(String accountId, String userId) {

Transaction transactionToUndo = undoRedoStack.popUndo(accountId);

if (transactionToUndo == null) {

throw new NoTransactionToUndoException("No transaction available to undo");

}

try {

// Reverse the transaction

Transaction reversedTransaction = reverseTransaction(transactionToUndo, userId);

// Log the undo operation

auditService.logSuccess(userId, "UNDO\_TRANSACTION", "TRANSACTION",

reversedTransaction.getId(),

"Transaction " + transactionToUndo.getTransactionId() + " undone successfully");

return reversedTransaction;

} catch (Exception e) {

// Push back to undo stack if reversal fails

undoRedoStack.pushUndo(accountId, transactionToUndo);

auditService.logFailure(userId, "UNDO\_TRANSACTION", "TRANSACTION",

transactionToUndo.getId(),

"Failed to undo transaction " + transactionToUndo.getTransactionId(),

e.getMessage());

throw e;

}

}

}

2. REDO TRANSACTION SERVICE

@Service

public class RedoTransactionService {

@Autowired

private UndoRedoStack undoRedoStack;

@Autowired

private AccountRepository accountRepository;

@Autowired

private TransactionRepository transactionRepository;

@Autowired

private AuditService auditService;

public Transaction redoLastTransaction(String accountId, String userId) {

Transaction transactionToRedo = undoRedoStack.popRedo(accountId);

if (transactionToRedo == null) {

throw new NoTransactionToRedoException("No transaction available to redo");

}

try {

// Re-execute the transaction

Transaction redoneTransaction = reexecuteTransaction(transactionToRedo, userId);

// Log the redo operation

auditService.logSuccess(userId, "REDO\_TRANSACTION", "TRANSACTION",

redoneTransaction.getId(),

"Transaction " + transactionToRedo.getTransactionId() + " redone successfully");

return redoneTransaction;

} catch (Exception e) {

// Push back to redo stack if re-execution fails

undoRedoStack.pushRedo(accountId, transactionToRedo);

auditService.logFailure(userId, "REDO\_TRANSACTION", "TRANSACTION",

transactionToRedo.getId(),

"Failed to redo transaction " + transactionToRedo.getTransactionId(),

e.getMessage());

throw e;

}

}

}

BATCH SETTLEMENT IMPLEMENTATION

-------------------------------

1. SETTLEMENT PROCESSOR

@Service

public class SettlementProcessor {

@Autowired

private SettlementQueue settlementQueue;

@Autowired

private TransactionRepository transactionRepository;

@Autowired

private AccountRepository accountRepository;

@Autowired

private AuditService auditService;

@Scheduled(fixedRate = 300000) // Run every 5 minutes

public void processSettlementBatch() {

List<Transaction> transactions = settlementQueue.drainTo(100); // Process up to 100 transactions

if (transactions.isEmpty()) {

return;

}

int processedCount = 0;

int failedCount = 0;

for (Transaction transaction : transactions) {

try {

processSettlementTransaction(transaction);

processedCount++;

} catch (Exception e) {

failedCount++;

log.error("Failed to process settlement transaction: {}", transaction.getTransactionId(), e);

}

}

// Log settlement results

auditService.logSuccess("SYSTEM", "BATCH\_SETTLEMENT", "SETTLEMENT", "BATCH",

"Settlement batch completed. Processed: " + processedCount + ", Failed: " + failedCount);

}

}

2. INDIVIDUAL TRANSACTION SETTLEMENT

private void processSettlementTransaction(Transaction transaction) {

// Validate transaction

if (transaction.getStatus() != TransactionStatus.PENDING) {

throw new InvalidTransactionStatusException("Transaction is not in PENDING status");

}

// Process based on transaction type

switch (transaction.getTransactionType()) {

case DEPOSIT:

processDepositSettlement(transaction);

break;

case WITHDRAWAL:

processWithdrawalSettlement(transaction);

break;

case TRANSFER:

processTransferSettlement(transaction);

break;

default:

throw new UnsupportedTransactionTypeException("Unsupported transaction type: " + transaction.getTransactionType());

}

// Update transaction status

transaction.setStatus(TransactionStatus.COMPLETED);

transaction.setProcessedDate(LocalDateTime.now());

transactionRepository.save(transaction);

}

THREAD SAFETY IMPLEMENTATION

----------------------------

1. CONCURRENT HASH MAP USAGE

- UndoRedoStack uses ConcurrentHashMap for thread-safe operations

- Multiple threads can safely access and modify stacks simultaneously

- Atomic operations prevent race conditions

- No external synchronization required

2. BLOCKING QUEUE IMPLEMENTATION

- SettlementQueue uses LinkedBlockingQueue for thread-safe operations

- Producer-consumer pattern implementation

- Automatic blocking when queue is full or empty

- Thread-safe enqueue and dequeue operations

3. SYNCHRONIZATION STRATEGIES

- Fine-grained locking for better performance

- Lock-free data structures where possible

- Atomic operations for simple state changes

- Thread-local storage for per-thread data

PERFORMANCE OPTIMIZATION

------------------------

1. MEMORY MANAGEMENT

- Efficient object creation and reuse

- Proper garbage collection optimization

- Memory leak prevention

- Resource cleanup in finally blocks

2. CACHING STRATEGIES

- Stack size caching for frequent queries

- Transaction object pooling

- Lazy initialization for expensive operations

- Cache invalidation strategies

3. SCALABILITY CONSIDERATIONS

- Horizontal scaling support

- Load balancing for queue processing

- Distributed data structure support

- Cluster-aware implementations

ERROR HANDLING AND RECOVERY

---------------------------

1. EXCEPTION HANDLING

- Custom exceptions for data structure operations

- Graceful error handling and recovery

- Transaction rollback on failures

- Comprehensive error logging

2. RECOVERY MECHANISMS

- Automatic retry for failed operations

- Dead letter queue for failed transactions

- State recovery after system failures

- Data consistency maintenance

3. MONITORING AND ALERTING

- Stack size monitoring

- Queue depth monitoring

- Performance metrics collection

- Alert generation for anomalies

TESTING IMPLEMENTATION

----------------------

1. UNIT TESTS

- Stack operations testing

- Queue operations testing

- Thread safety testing

- Edge case testing

2. INTEGRATION TESTS

- End-to-end transaction reversal testing

- Batch settlement testing

- Concurrent access testing

- Performance testing

3. LOAD TESTING

- High-volume transaction processing

- Concurrent user simulation

- Memory usage testing

- Performance benchmarking

This data structures implementation provides efficient transaction management capabilities with proper thread safety, error handling, and performance optimization.