TEAM ACHA

Harshit Singh (22CS10031)
Chandransh Singh(22CS30017)
Aryan Sanghi (22CS30013)
Arpit Sodhani (22CS30012)

15th April 2025

1 Problem Statement

Implementing a Robust Concurrency Manager with Two-Phase Locking and Deadlock Handling

1.1 Introduction

In modern multi-user systems such as database management systems, operating systems, and distributed applications, multiple processes or threads often need to access shared resources concurrently. Without proper coordination, this concurrent access can lead to race conditions, data inconsistencies, and system failures. To ensure correctness while maximizing parallelism, a robust concurrency control mechanism is essential. This project focuses on designing a Concurrency Manager responsible for coordinating access to shared resources using the Two-Phase Locking (2PL) protocol along with an effective deadlock detection and resolution mechanism.

1.2 Problem Overview

Design and implement a concurrency control system that enforces the Two-Phase Locking (2PL) protocol to manage transactional access to shared resources. The system must handle lock compatibility, transaction states, and waiting mechanisms when conflicts arise. Additionally, it should detect and resolve deadlocks that may occur due to resource contention. The design should support concurrent transactions and provide transparency through logging of system behavior.

1.3 Functional Expectations

- Ensure that transactions progress through appropriate lifecycle states.
- Enforce locking rules based on lock types and compatibility.
- Detect and resolve deadlocks through periodic analysis of transaction dependencies.
- Provide visibility into internal behavior through structured logging.

- Ensure thread-safe handling of all shared data structures to support concurrency.

1.4 Intricacies and Worthiness

This problem is non-trivial due to the inherent complexities of managing concurrency:

- Correctness vs. Performance: Maintaining correctness while maximizing concurrency introduces trade-offs. Locking may reduce throughput, and tuning is necessary.
- Deadlock Handling Overhead: Detecting and resolving deadlocks can be costly. Strategies for detection frequency and victim selection must be chosen carefully.
- Robustness: Edge cases such as transaction aborts during wait periods must be handled gracefully.
- Synchronization Complexity: Designing a thread-safe system without introducing additional synchronization-related issues requires precise coordination.

2 Methodology

This project implements a Concurrency Manager built around the Strict Two-Phase Locking (Strict 2PL) protocol, complemented with a proactive deadlock detection and resolution mechanism, structured through a Wait-For Graph (WFG) model. The architecture is carefully modularized, with responsibilities delegated to distinct C++ classes, enabling clean separation of concerns, extensibility, and maintainability.

2.1 Concurrency Control: Strict Two-Phase Locking (2PL)

The system ensures serializability by adhering to the Strict 2PL protocol, which dictates that a transaction acquires all necessary locks during its growing phase and releases them only during the shrinking phase. This prevents cascading aborts and ensures isolation across concurrent transactions.

The lock acquisition process is centralized in a Lock Manager, which maintains a structured mapping of resources to lock request queues. Each resource maintains an ordered list of lock requests — either granted or pending — with access regulated by a lock compatibility matrix. Shared locks allow concurrent access, while exclusive locks require complete isolation. This compatibility logic governs lock granting decisions at runtime.

Each transaction is associated with a Transaction object, which tracks its current state (GROWING, SHRINKING, COMMITTED, ABORTED), its held locks, and metadata like transaction priority and start time. The system strictly enforces state transitions: once a transaction begins releasing any lock, it enters the SHRINKING phase and cannot request new locks thereafter.

2.2 Class Structure Overview

The system comprises the following core classes, each playing a critical role in concurrency control:

1. Transaction

    - Each transaction maintains a unique ID, a priority level used for deadlock resolution, and a set of resources it currently holds locks on.
    - It operates as a stateful entity with a defined lifecycle. Internally, it transitions through an enum-like state machine representing the phases of the Strict 2PL protocol.
    - A static registry of all active transactions allows global access for inter-module coordination, such as during lock operations or deadlock detection.
    - It also tracks the lock state of each resource it interacts with, ensuring precisecontrol over lock ownership and lifecycle transitions.

2. Lock Manager

    - Manages resource-to-lock mappings through a central lock table (typically a map of resource IDs to lock request lists).
    - Handles both synchronous lock granting and blocking wait queues via condition variables.
    - Enforces lock compatibility and performs inline lock upgrades when possible (e.g., shared to exclusive).
    - Employs mutex protection over its internal data structures to ensure thread-safe access and modifications.
    - Integrates directly with the deadlock subsystem by signaling when a transaction must wait on another, thus updating the WFG indirectly.

3. Resource Allocation Graph / Wait-For Graph

    - The Resource Allocation Graph (RAG) maintains a bipartite graph of transactions and resources, capturing ownership and pending request edges.
    - For deadlock analysis, an explicit Wait-For Graph (WFG) is constructed periodically. It is a simplified view, mapping transactions to other transactions they are blocked by.
    - The WFG is built by scanning the lock table and identifying all transactions waiting on resources held by others.
    - Cycles in this graph are indicative of deadlocks, and the cycle detection algorithm focuses on identifying minimal-priority victims.

4. Deadlock Detector

    - Operates on a dedicated thread, periodically scanning the WFG for cycles using depth-first search with backtracking.
    - The detection strategy identifies a victim transaction using a priority-based heuristic — the lowest-priority transaction in a detected cycle is aborted to break the deadlock.
    - The detector is decoupled from the core LockManager logic, instead querying its internal state to build a current snapshot of dependencies.

5. Logger

- Provides structured logging with timestamped entries and log-level filtering(DEBUG, INFO, WARNING, ERROR, FATAL).
- Ensures thread-safe log access and optionally mirrors output to the console.
- Specialized log entries track events like lock acquisition, transaction lifecycle changes, and deadlock detection/resolution, helping in debugging and traceability.

6. Concurrency Manager

    - Serves as the orchestrator and main public API, exposing methods to start, commit, or abort transactions and to acquire or release locks.
    - Internally maintains ownership over all subsystem instances (LockManager, DeadlockDetector, Logger, etc.), managing their life cycles and interactions.
    - Encapsulates transaction management, from assignment of transaction IDs to releasing all associated locks upon termination (commit or abort).
    - Coordinates with the LockManager and DeadlockDetector to maintain system consistency and responsiveness in high-contention scenarios.

2.3 Deadlock Handling and Recovery

The most complex challenge addressed in this design is the handling of deadlocks, which naturally emerge in locking-based systems. The system detects potential deadlocks via periodic graph analysis of the WFG, built from current lock holder/waiter relationships. When a cycle is found, the system selects a victim transaction using its stored priority values, preferring to abort the lowest-priority one. Upon abortion, the victim's locks are released, and other transactions are notified to reattempt acquisition. This mechanism guarantees forward progress and prevents indefinite blocking(livelock).

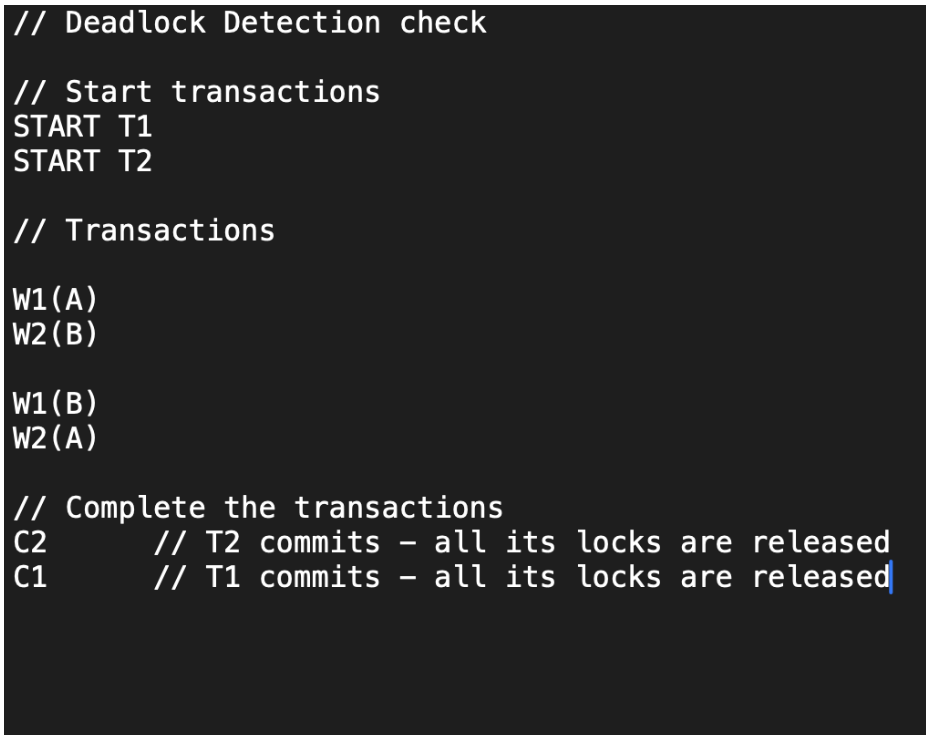
2.4 Testing and Validation

A dedicated test framework simulates multiple concurrent transactions through a test harness with defined execution traces. It interprets scripted transaction operations(READ, WRITE, COMMIT, etc.) and translates them into lock requests and transaction commands executed concurrently via threads. The test system:

- Assigns transactions to threads.
- Logs intermediate states (e.g., resource graph snapshots).
- Validates serializability and freedom from deadlocks.
- Emulates real-world timing variations using delays and sleeps between operations to encourage interleaving.

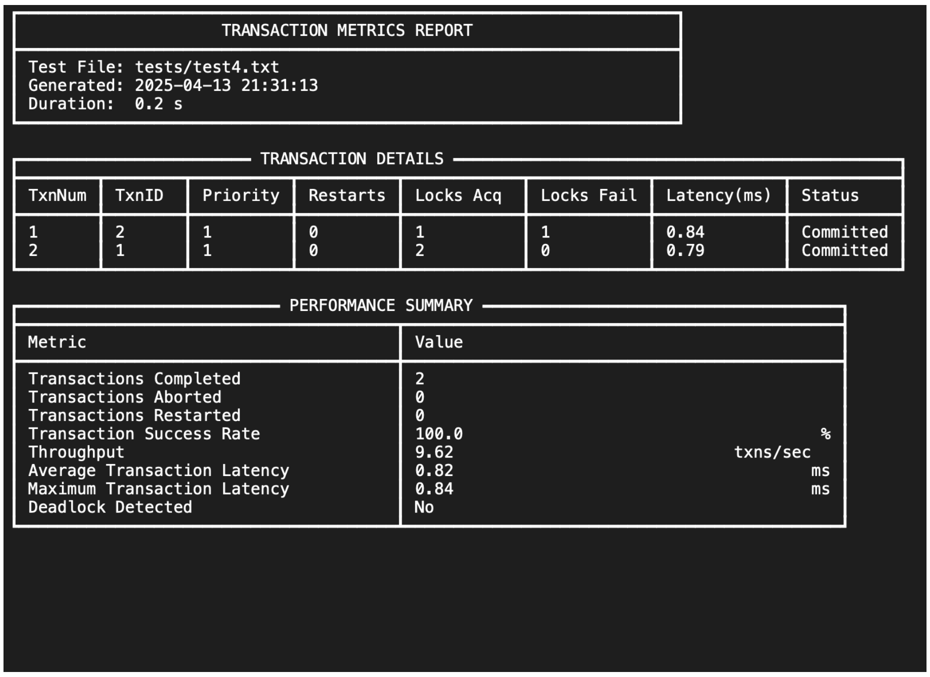
3 Result and screenshots

3.1 Small Test



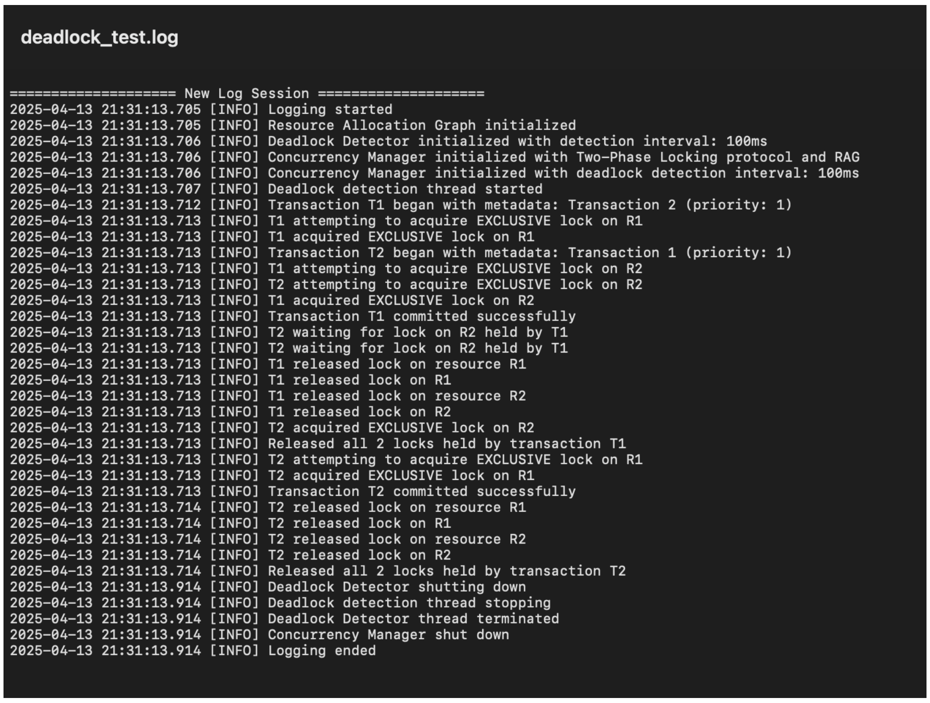
```
// Deadlock Detection check

// Start transactions
START T1
START T2

// Transactions

W1(A)
W2(B)

W1(B)
W2(A)

// Complete the transactions
C2      // T2 commits - all its locks are released
C1      // T1 commits - all its locks are released
```

Figure 1: Sample test case



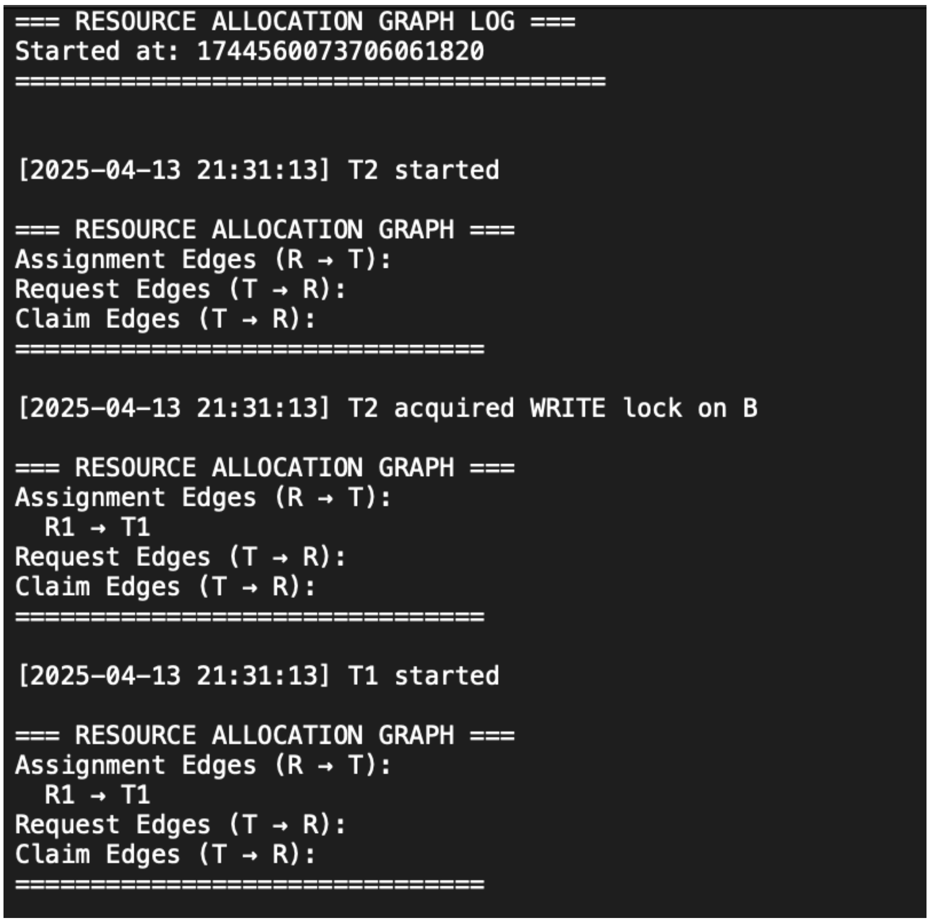
```
                TRANSACTION METRICS REPORT

 Test File: tests/test4.txt
 Generated: 2025-04-13 21:31:13
 Duration:  0.2 s

                      TRANSACTION DETAILS
 TxnNum | TxnID | Priority | Restarts | Locks Acq | Locks Fail | Latency(ms) | Status
 1      | 2     | 1        | 0        | 1         | 1          | 0.84        | Committed
 2      | 1     | 1        | 0        | 2         | 0          | 0.79        | Committed

                      PERFORMANCE SUMMARY
 Metric                        | Value
 Transactions Completed        | 2
 Transactions Aborted          | 0
 Transactions Restarted        | 0
 Transaction Success Rate      | 100.0                    %
 Throughput                    | 9.62              txns/sec
 Average Transaction Latency   | 0.82                    ms
 Maximum Transaction Latency   | 0.84                    ms
 Deadlock Detected             | No
```

Figure 2: Transactions



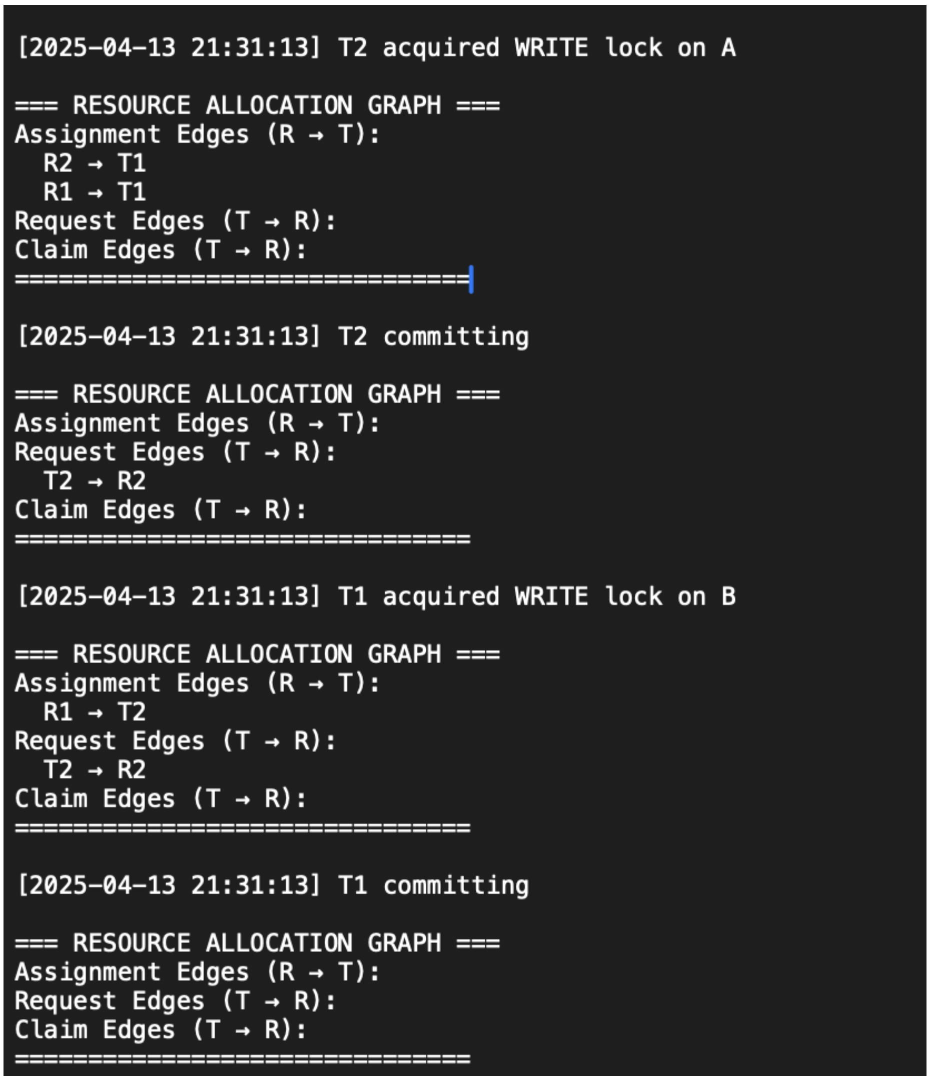
```
deadlock_test.log

==================== New Log Session ====================
2025-04-13 21:31:13.705 [INFO] Logging started
2025-04-13 21:31:13.705 [INFO] Resource Allocation Graph initialized
2025-04-13 21:31:13.706 [INFO] Deadlock Detector initialized with detection interval: 100ms
2025-04-13 21:31:13.706 [INFO] Concurrency Manager initialized with Two-Phase Locking protocol and RAG
2025-04-13 21:31:13.706 [INFO] Concurrency Manager initialized with deadlock detection interval: 100ms
2025-04-13 21:31:13.707 [INFO] Deadlock detection thread started
2025-04-13 21:31:13.712 [INFO] Transaction T1 began with metadata: Transaction 2 (priority: 1)
2025-04-13 21:31:13.713 [INFO] T1 attempting to acquire EXCLUSIVE lock on R1
2025-04-13 21:31:13.713 [INFO] T1 acquired EXCLUSIVE lock on R1
2025-04-13 21:31:13.713 [INFO] Transaction T2 began with metadata: Transaction 1 (priority: 1)
2025-04-13 21:31:13.713 [INFO] T1 attempting to acquire EXCLUSIVE lock on R2
2025-04-13 21:31:13.713 [INFO] T2 attempting to acquire EXCLUSIVE lock on R2
2025-04-13 21:31:13.713 [INFO] T1 acquired EXCLUSIVE lock on R2
2025-04-13 21:31:13.713 [INFO] Transaction T1 committed successfully
2025-04-13 21:31:13.713 [INFO] T2 waiting for lock on R2 held by T1
2025-04-13 21:31:13.713 [INFO] T2 waiting for lock on R2 held by T1
2025-04-13 21:31:13.713 [INFO] T1 released lock on resource R1
2025-04-13 21:31:13.713 [INFO] T1 released lock on R1
2025-04-13 21:31:13.713 [INFO] T1 released lock on resource R2
2025-04-13 21:31:13.713 [INFO] T1 released lock on R2
2025-04-13 21:31:13.713 [INFO] T2 acquired EXCLUSIVE lock on R2
2025-04-13 21:31:13.713 [INFO] Released all 2 locks held by transaction T1
2025-04-13 21:31:13.713 [INFO] T2 attempting to acquire EXCLUSIVE lock on R1
2025-04-13 21:31:13.713 [INFO] T2 acquired EXCLUSIVE lock on R1
2025-04-13 21:31:13.713 [INFO] Transaction T2 committed successfully
2025-04-13 21:31:13.714 [INFO] T2 released lock on resource R1
2025-04-13 21:31:13.714 [INFO] T2 released lock on R1
2025-04-13 21:31:13.714 [INFO] T2 released lock on resource R2
2025-04-13 21:31:13.714 [INFO] T2 released lock on R2
2025-04-13 21:31:13.714 [INFO] Released all 2 locks held by transaction T2
2025-04-13 21:31:13.914 [INFO] Deadlock Detector shutting down
2025-04-13 21:31:13.914 [INFO] Deadlock detection thread stopping
2025-04-13 21:31:13.914 [INFO] Deadlock Detector thread terminated
2025-04-13 21:31:13.914 [INFO] Concurrency Manager shut down
2025-04-13 21:31:13.914 [INFO] Logging ended
```

Figure 3: Deadlock



```
=== RESOURCE ALLOCATION GRAPH LOG ===
Started at: 1744560073706061820
=======================================


[2025-04-13 21:31:13] T2 started

=== RESOURCE ALLOCATION GRAPH ===
Assignment Edges (R → T):
Request Edges (T → R):
Claim Edges (T → R):
===============================

[2025-04-13 21:31:13] T2 acquired WRITE lock on B

=== RESOURCE ALLOCATION GRAPH ===
Assignment Edges (R → T):
  R1 → T1
Request Edges (T → R):
Claim Edges (T → R):
===============================

[2025-04-13 21:31:13] T1 started

=== RESOURCE ALLOCATION GRAPH ===
Assignment Edges (R → T):
  R1 → T1
Request Edges (T → R):
Claim Edges (T → R):
===============================
```

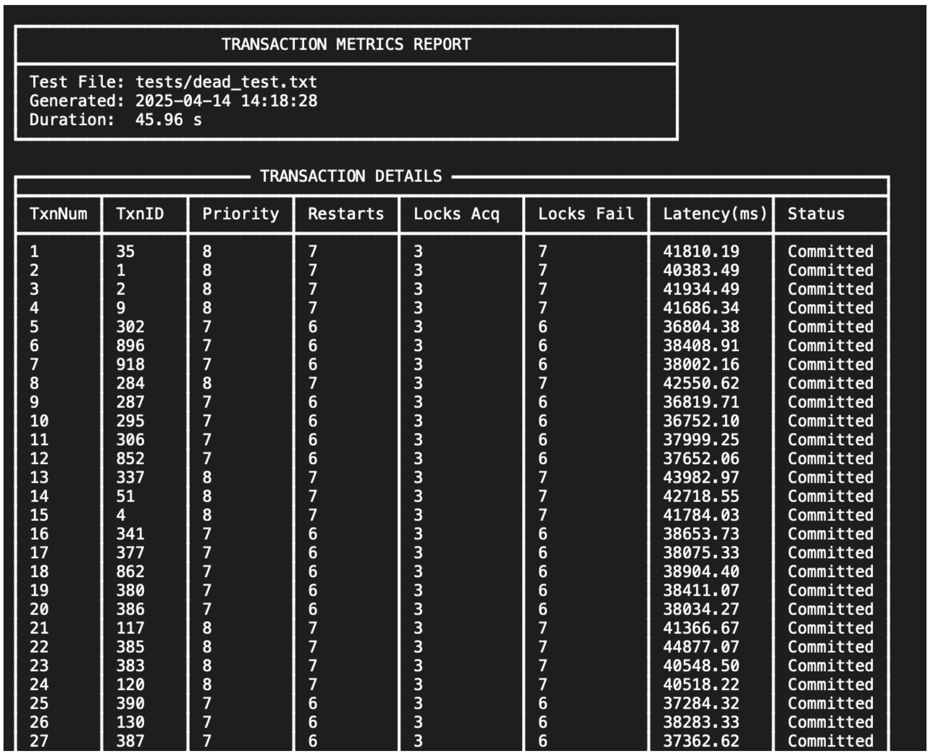
Figure 4: Resource Allocation Graph - 1



```
[2025-04-13 21:31:13] T2 acquired WRITE lock on A

=== RESOURCE ALLOCATION GRAPH ===
Assignment Edges (R → T):
  R2 → T1
  R1 → T1
Request Edges (T → R):
Claim Edges (T → R):
===============================

[2025-04-13 21:31:13] T2 committing

=== RESOURCE ALLOCATION GRAPH ===
Assignment Edges (R → T):
Request Edges (T → R):
  T2 → R2
Claim Edges (T → R):
===============================

[2025-04-13 21:31:13] T1 acquired WRITE lock on B

=== RESOURCE ALLOCATION GRAPH ===
Assignment Edges (R → T):
  R1 → T2
Request Edges (T → R):
  T2 → R2
Claim Edges (T → R):
===============================

[2025-04-13 21:31:13] T1 committing

=== RESOURCE ALLOCATION GRAPH ===
Assignment Edges (R → T):
Request Edges (T → R):
Claim Edges (T → R):
===============================
```

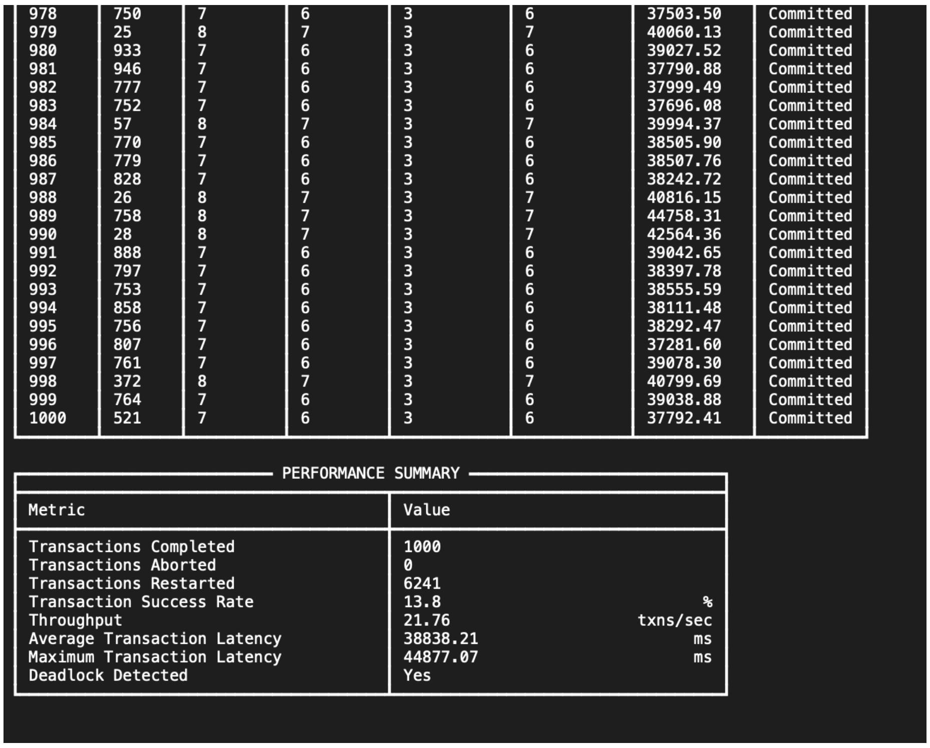
Figure 5: Resource Allocation Graph - 2

3.2 Large Test



```
TRANSACTION METRICS REPORT

Test File: tests/dead_test.txt
Generated: 2025-04-14 14:18:28
Duration:  45.96 s

TRANSACTION DETAILS

TxnNum  TxnID  Priority  Restarts  Locks Acq  Locks Fail  Latency(ms)  Status

1       35     8         7         3          7           41810.19     Committed
2       1      8         7         3          7           40383.49     Committed
3       2      8         7         3          7           41934.49     Committed
4       9      8         7         3          7           41686.34     Committed
5       302    7         6         3          6           36804.38     Committed
6       896    7         6         3          6           38408.91     Committed
7       918    7         6         3          6           38002.16     Committed
8       284    8         7         3          7           42550.62     Committed
9       287    7         6         3          6           36819.71     Committed
10      295    7         6         3          6           36752.10     Committed
11      306    7         6         3          6           37999.25     Committed
12      852    7         6         3          6           37652.06     Committed
13      337    8         7         3          7           43982.97     Committed
14      51     8         7         3          7           42718.55     Committed
15      4      8         7         3          7           41784.03     Committed
16      341    7         6         3          6           38653.73     Committed
17      377    7         6         3          6           38075.33     Committed
18      862    7         6         3          6           38904.40     Committed
19      380    7         6         3          6           38411.07     Committed
20      386    7         6         3          6           38034.27     Committed
21      117    8         7         3          7           41366.67     Committed
22      385    8         7         3          7           44877.07     Committed
23      383    8         7         3          7           40548.50     Committed
24      120    8         7         3          7           40518.22     Committed
25      390    7         6         3          6           37284.32     Committed
26      130    7         6         3          6           38283.33     Committed
27      387    7         6         3          6           37362.62     Committed
```

Figure 6: Transactions - 1



```
978     750    7         6         3          6           37503.50     Committed
979     25     8         7         3          7           40060.13     Committed
980     933    7         6         3          6           39027.52     Committed
981     946    7         6         3          6           37790.88     Committed
982     777    7         6         3          6           37999.49     Committed
983     752    7         6         3          6           37696.08     Committed
984     57     8         7         3          7           39994.37     Committed
985     770    7         6         3          6           38505.90     Committed
986     779    7         6         3          6           38507.76     Committed
987     828    7         6         3          6           38242.72     Committed
988     26     8         7         3          7           40816.15     Committed
989     758    8         7         3          7           44758.31     Committed
990     28     8         7         3          7           42564.36     Committed
991     888    7         6         3          6           39042.65     Committed
992     797    7         6         3          6           38397.78     Committed
993     753    7         6         3          6           38555.59     Committed
994     858    7         6         3          6           38111.48     Committed
995     756    7         6         3          6           38292.47     Committed
996     807    7         6         3          6           37281.60     Committed
997     761    7         6         3          6           39078.30     Committed
998     372    8         7         3          7           40799.69     Committed
999     764    7         6         3          6           39038.88     Committed
1000    521    7         6         3          6           37792.41     Committed

PERFORMANCE SUMMARY

Metric                        Value

Transactions Completed        1000
Transactions Aborted          0
Transactions Restarted        6241
Transaction Success Rate      13.8                     %
Throughput                    21.76             txns/sec
Average Transaction Latency   38838.21                ms
Maximum Transaction Latency   44877.07                ms
Deadlock Detected             Yes
```

Figure 7: Transactions - 2

4 Future scope:

4.1 Deadlock Handling Enhancements

- **Adaptive Detection:** Adjust deadlock detection frequency based on system load or deadlock frequency.
- **Smarter Victim Selection:** Incorporate transaction age, rollback cost, or number of locks held into the victim selection logic beyond just priority.
- **Prevention Strategies:** Explore implementing deadlock prevention schemes like Wait-Die or Wound-Wait as alternatives or complements.

4.2 Alternative Concurrency Protocols

- **MVCC(Multi-Version Concurrency Control):** Implement MVCC to allow readers and writers to operate concurrently without blocking each other, potentially offering higher throughput for read-heavy workloads.
- **OCC(Optimistic Concurrency Control):** Explore OCC for scenarios where conflicts are rare, avoiding locking overhead but requiring validation and potential rollbacks at commit time.

4.3 Performance & Scalability

- **Data Structures:** Optimize internal data structures (e.g., lock table hashing, WFG representation).
- **Concurrency:** Refine mutex usage and condition variable signaling ('notify_one' where applicable) to reduce contention.

4.4 Recovery & Durability

- **Write-Ahead Logging (WAL):** Implement robust logging (like ARIES) to ensure transaction atomicity and durability across system crashes.
- **Checkpointing & Recovery:** Add mechanisms for checkpointing and system recovery after failures.

5 Conclusion

The implemented methodology successfully addresses the challenge of concurrency control by combining the widely accepted Strict 2PL protocol with a proactive deadlock detection and priority-based resolution mechanism. The modular design separates concerns into distinct components (Transaction, LockManager, DeadlockDetector, Logger), enhancing maintainability and testability. Thread safety is addressed through mutexes and condition variables. The explicit WFG construction and periodic cycle detection provide a robust solution to deadlocks inherent in locking protocols. Comprehensive logging offers vital visibility into the system's dynamic behavior. The test runner validates the implementation against simulated concurrent workloads. This approach provides a strong foundation for ensuring data consistency and system liveness in a multi-threaded environment.

6 References

- Silberschatz, Korth & Sudarshan – Database System Concepts (7th Edition)
- Bernstein, Hadzilacos & Goodman – Concurrency Control and Recovery in Database Systems
- CMU 15-445/645: Intro to Database Systems – Project 4: Concurrency Control `https://15445.courses.cs.cmu.edu/fall2020/project4`
- Lamport, Leslie – Time, Clocks, and the Ordering of Events in a Distributed System[Research Paper] `https://lamport.azurewebsites.net/pubs/time-clocks.pdf`
- Microsoft Docs – Thread Synchronization in C++ `https://learn.microsoft.com/enus/windows/win32/sync/synchronization`