# GEORGIA INSTITUTE OF TECHNOLOGY CS 6423: DATABASE SYSTEM IMPLEMENTATION PART II (SPRING 2025)

Exercise Sheet 1
Version 1.1
Due: Feb 20, 2025 @ 11:59pm

February 21, 2025

## Answer Sheet Instructions

- Write all answers in the designated spaces below
- Use bullet points for brief answers where appropriate
- Maintain original question numbering
- Submit completed PDF to Gradescope

## Scoring Overview

| Question | Points | Score |
|---|---|---|
| Recovery Protocol | 40 | |
| Logging Protocol | 70 | |
| SiloR protocol | 20 | |
| Total: | 130 | |

## Question 1: Recovery Protocol Answers

### (a) ARIES Recovery Analysis [40 points]

#### i. ATT & DPT Tables [20 points]

**Active Transaction Table (ATT):**

| Txn ID | LastLSN | Status |
|---|---|---|
| $T1$ | 12 | U |
| $T4$ | 20 | C |

**Dirty Page Table (DPT):**

| Page ID | recLSN |
|---|---|
| $P1$ | 6 |
| $P2$ | 2 |
| $P3$ | 12 |
| $P4$ | 5 |
| $P5$ | 14 |
| $P6$ | 17 |

**ii. Redo/Undo Log Records [20 points]**

- For Redo Phase: We add one `TXN-END` (reference)

| LSN | prevLSN | TxnId | Type | PageID | Object | Before | After | UndoNext |
|---|---|---|---|---|---|---|---|---|
| 24 | 21 | T4 | TXN-END | - | - | - | - | nil |

Table 1: Redo Log Sequence Table

- For Undo Phase:

| LSN | prevLSN | TxnId | Type | PageID | Object | Before | After | UndoNext |
|---|---|---|---|---|---|---|---|---|
| 25 | 12 | T1 | CLR | P3 | A | 11 | 10 | 9 |
| 26 | 25 | T1 | CLR | P1 | B | 29 | 19 | 8 |
| 27 | 26 | T1 | CLR | P1 | U | 40 | 35 | 7 |
| 28 | 27 | T1 | CLR | P1 | V | 35 | 30 | 6 |
| 29 | 28 | T1 | CLR | P1 | W | 21 | 12 | 4 |
| 30 | 29 | T1 | TXN-END | - | - | - | - | nil |
| 31 | 22 | T3 | TXN-END | - | - | - | - | nil |

Table 2: Undo Log Sequence Table

The before and after are reversed in undo log.

# Question 2: Logging Protocol Answers [70 points]

## (a) Shadow Paging

### i. B+-Tree Sharing [10 points]

In a B+-tree with copy-on-write semantics, updates propagate from leaves to the root along the affected path:

- Leaf updates create new leaf nodes while preserving original parent pointers. We have no need to copy entire tree, only need to copy paths in the tree that lead to updated leaf nodes (ref: here)
- Internal-node propagation copies only ancestors directly referencing modified leaves. For example, updating leaf L3 in a 3-level (Fig 1) tree requires copying:
  - L3 (Leaf)
  - Its parent P1
  - The root R
- Unmodified subtrees (e.g., siblings of P1 or unrelated branches) remain shared between master and shadow copies. This selective copying ensures structural consistency without duplicating unaffected nodes

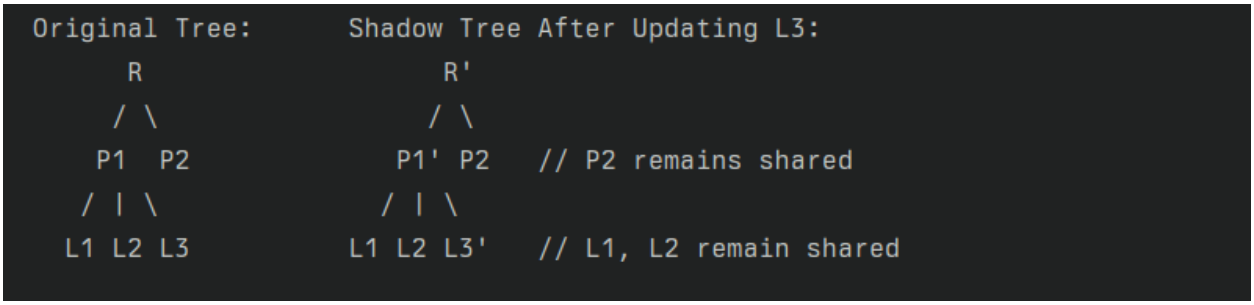


Figure 1: B+Tree Example - Shadow Paging

### ii. WAL vs Shadow [10 points]

Write-ahead logging is generally faster for small updates because it minimizes the number of writes, whereas shadow paging incurs higher overhead due to node copying. However, shadow paging has advantages in crash recovery, as it eliminates the need for redo/undo logging by maintaining a consistent tree structure at all times.

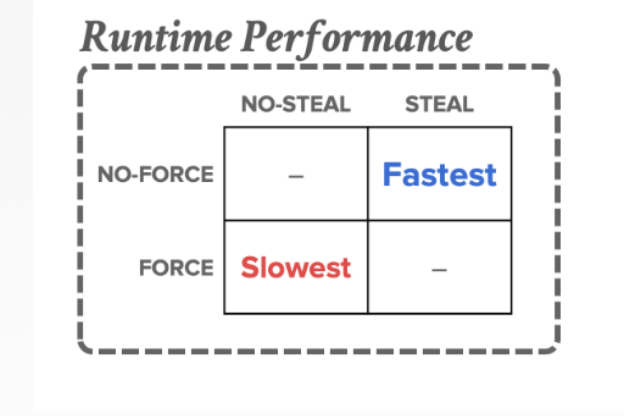


Figure 2: Runtime Performance

## (b) Physiological Logging [10 points]

Physiological logging is a hybrid approach where log records target a specific page but do not specify the exact byte-level changes. This makes it more efficient than physical logging, which logs exact modifications at the byte or block level (reference)

As explained in class, take the example of updating a tuple in the page. Let us assume that page contains 100 tuples, each 100 bytes in size. A new txn updates a single tuple in the page. As compared to physical logging, which records the exact byte change the by logging the entire page (4 KB) physiological logging logs only the operation on the specific tuple within the page. Instead of storing entire page, the log only contains PageID, TupleID, and Before/After values. The log entry size is significantly less compared to 4KB.

## (c) Buffer Policies [30 points]

### i. NO-STEAL + FORCE

- **Policy**:
  - Uncommitted changes remain in memory (NO-STEAL)
  - Committed changes immediately written to disk (FORCE)
- **Logging**:
  - Neither undo nor redo required
- **Recovery**:
  - Trivial - no disk cleanup needed
- **Trade-off**:
  - High commit I/O, memory-bound transactions

### ii. STEAL + NO-FORCE

- **Policy**:
  - Uncommitted changes may write to disk (STEAL)
  - Committed changes may remain in memory (NO-FORCE)
- **Logging**:
  - Both undo (for aborts) and redo (for commits) required
- **Recovery**:
  - Complex two-phase process
- **Trade-off**:
  - Optimal runtime, complex logging

## iii. NO-STEAL + NO-FORCE

- **Policy**:
  - Uncommitted changes stay in memory (NO-STEAL)
  - Committed changes may stay in memory (NO-FORCE)
- **Logging**:
  - Redo required, no undo needed
- **Recovery**:
  - Replay committed operations
- **Trade-off**:
  - Memory-intensive, moderate recovery

## (d) Logical Logging [10 points]

Logical logging records high-level operations instead of specific physical changes (they do not keep track of which pages are actually modified). While the logging is significantly easier, it has its own set of challenges:

- Since the logical logs only store the operations they are conducting, determining the exact state of DB before crash is difficult, especially if there are concurrent transactions running
  - Consider the following case:

    ```
    Txn 1:  UPDATE accounts SET balance = balance - 100 WHERE id = 1
    Txn 2:  UPDATE accounts SET balance = balance + 50 WHERE id = 1
    ```

  - If a crash occurs mid-way, it's unclear what the balance was at different times
- Logical logging requires re-executing transactions during recovery and if concurrent transactions interleaved their updates, re-executing them in a different order may lead to incorrect results which violates the Isolation property as we learnt in class
  - `Txn 1:  INSERT INTO orders VALUES (1, 'itemA')`
  - `Txn 2:  INSERT INTO orders VALUES (2, 'itemB')`
  - If re-executed in the wrong order, Txn 2 might commit first, altering dependencies
- Some transactions depend on external factors
  - `Txn 1:  DELETE FROM users WHERE last_login < NOW() - INTERVAL '1 year'`
  - The result might be different when replayed later in time
- Undoing a logical operation is difficult because queries can have cascading effects
  - `Txn 1:  DELETE FROM orders WHERE amount < 10`
  - If rolled back, which exact rows should be restored? The deleted rows may not be explicitly stored

# Question 3: SiloR Protocol Answers [20 points]

## (a) Log Replay [10 points]

SiloR achieves high-speed log replay through **parallelism** and **value logging**:

- **Parallel I/O:**
  - Logs distributed across multiple disks with dedicated logger threads
  - 8 threads per disk process logs concurrently in reverse epoch order
  - Maximizes disk bandwidth utilization
- **Value Logging:**
  - Logs contain final committed values (not operations)
  - Enables out-of-order replay using TID comparisons
  - Latest TID per key automatically determines correct value
- **Epoch-Based Grouping:**
  - Processes logs in epoch units to avoid cross-epoch dependencies
  - Skips epochs beyond last persistent epoch (`pepoch`)

## (b) Value Logging [10 points]

Value logging works because:

- **TID Structure:**
  - Transaction IDs embed epoch numbers and worker-specific counters
  - Higher TIDs reflect later writes in serial order
- **Idempotent Replay:**
  - Higher TID values overwrite previous versions
  - Final state depends only on maximum TID per key
- **Epoch Boundaries:**
  - Anti-dependencies resolved at epoch granularity
  - TID ordering ensures correctness within epochs