CS 186 Exam Prep 4
Transactions & Concurrency, Recovery

Agenda

I. 6:10-6:30 -- Mini-lecture
II. 6:30-7:00 -- Go over answers
III. 7:00-8:00 -- Conceptual OH

Worksheet can be found at @471 on Piazza. Slides can be found on the same Piazza post.

Transactions & Concurrency

Why Transactions?

- Usually have multiple users accessing the database concurrently
- Can cause these problems:
- **Inconsistent Reads:** A user reads only part of what was updated (one user updates two tables, another user reads old version of one table and new version of the other table)
- **Lost Update:** Two users try to update the same record so one of the updates gets lost
- **Dirty Reads:** One user reads an update that was never committed (usually due to reading after abort but before rollback)

Transactions

- A sequence of multiple actions that should be executed as a single, logical, atomic unit. Abbreviated as “Xact”. Enforces these properties:
- **Atomicity**: A transaction ends in two ways: it either commits or aborts; either all actions in the Xact happen, or none happen.
- **Consistency**: If the DB starts out consistent (adhering to all rules), it ends up consistent at the end of the Xact.
- **Isolation**: Execution of each Xact is isolated from that of others; DBMS will ensure that each Xact executes as if it ran by itself, even with interleaved actions
- **Durability**: If a Xact commits, its effects persist; the effects of a committed Xact must survive failures.

Equivalence and Serializability

- Easiest way to enforce Isolation is to run transactions one at a time (a serial schedule), but this is inefficient
- Two schedules are **equivalent** if
  - They involve the same transactions
  - Each transaction has its operations in the same order
  - The final state after all the transactions is the same
- If a schedule is equivalent to a serial schedule, it is **serializable**
- Some schedules that interleave transaction actions are serializable, but it's hard to check.

Conflict Serializability

- Two operations in a schedule **conflict** if:
  - at least one operation is a write
  - they are on *different* transactions
  - they work on the *same* resource
- Conflicts are basically just pairs of operations that we need to be careful about

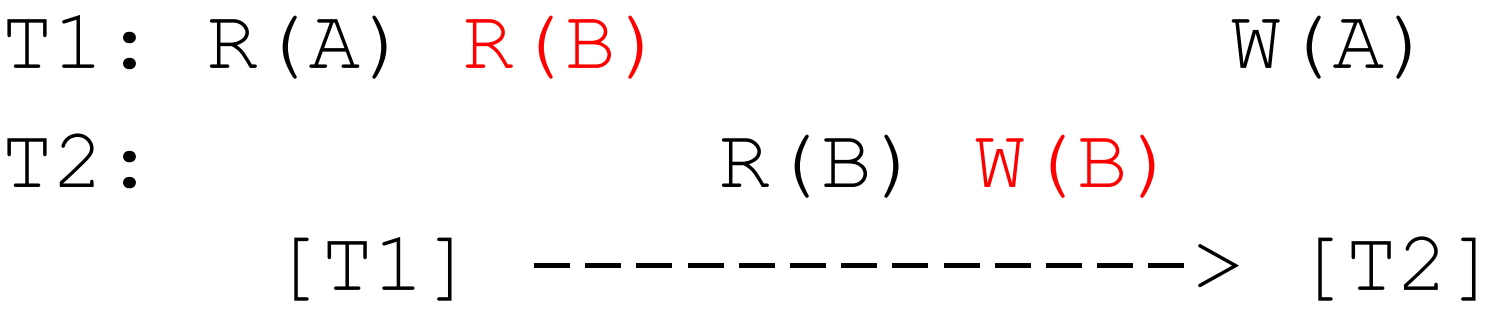
```
T1: R(A) R(B)                 W(A)
T2:              R(B) W(B)
```

Conflict Serializability

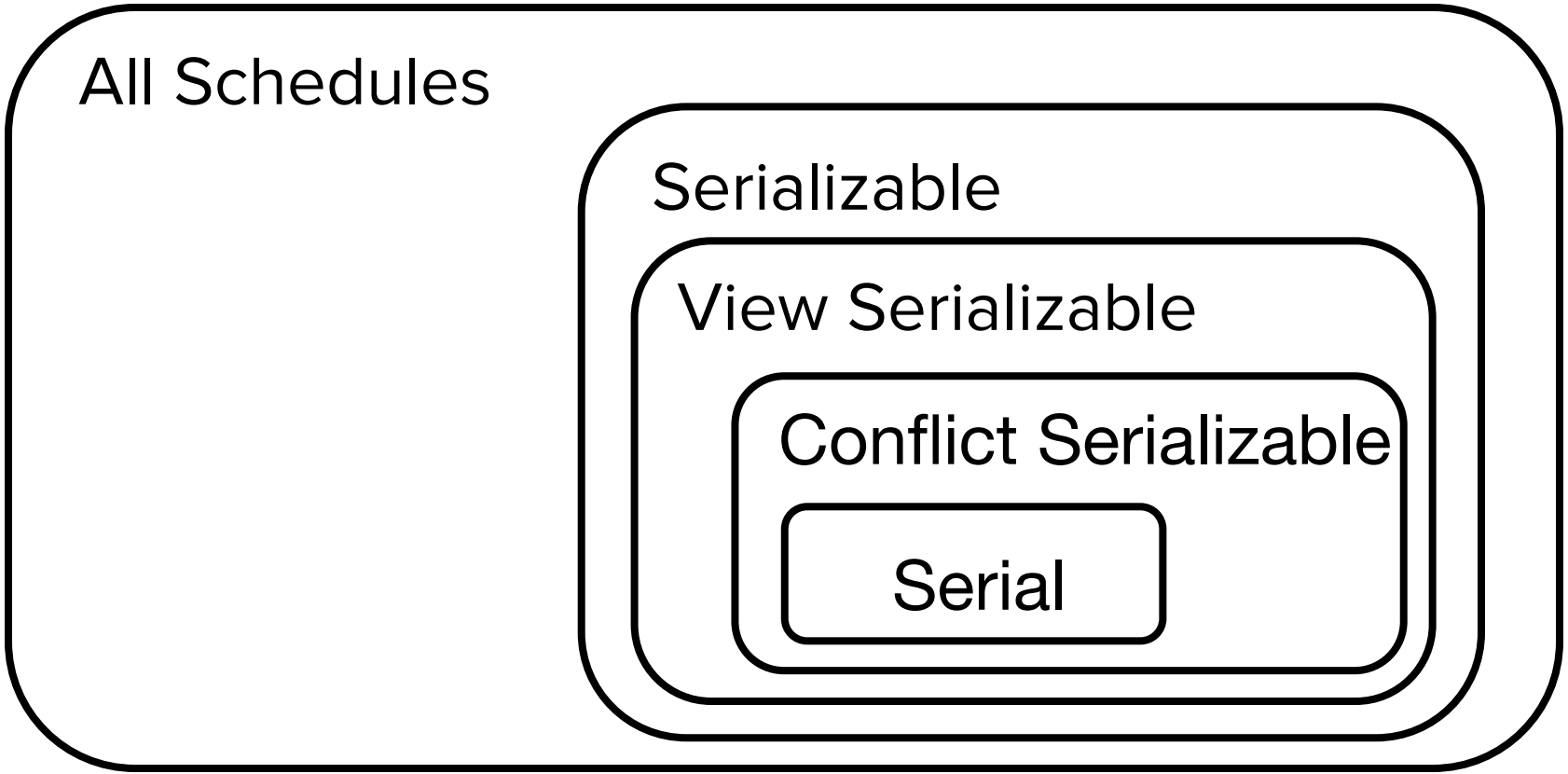
- If two schedules order their conflicting pairs the same way, they are **conflict equivalent** (and thus equivalent).
- If a schedule is conflict equivalent to a serial schedule, it is **conflict serializable**.
- Conflict serializability is a more strict condition than serializability (all conflict serializable schedules are serializable, but not all serializable schedules are conflict serializable). However, it's a lot easier to check.
- View equivalence/serializability falls in between them in terms of difficulty, but it's NP hard to check for.
    - Essentially, check same conditions as conflict serializability, except you can ignore blind writes (two writes without an interleaving read)

Conflict Serializability

- How do we check for conflict equivalence/serializability?
  - We build a **dependency graph** (**precedence graph**)
    - If an operation in T_i conflicts with an operation in T_j, and the operation in T_i comes first, add an edge from T_i to T_j
    - Cycle ➜ not conflict serializable



Types of Serializability

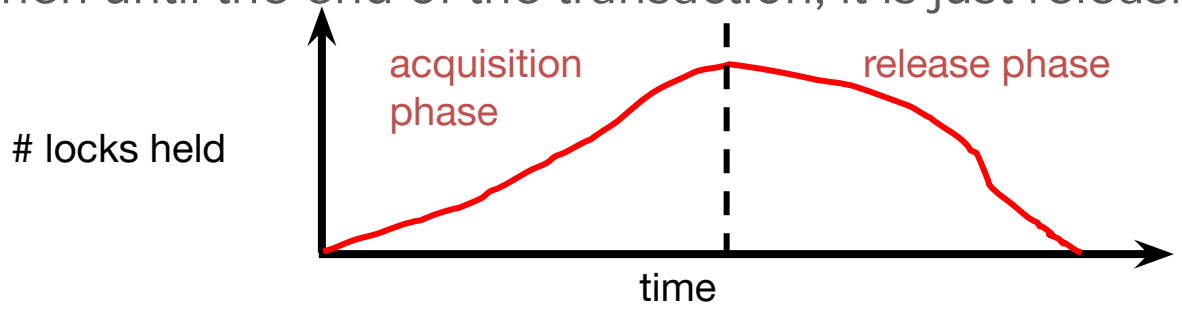


Locks

- Make sure that no other transaction is modifying the resource while you are using that resource
- Lock types: for a given resource A,
  - **S (Shared)** can read A and all descendants of A.
  - **X (Exclusive)** can read and write A and all descendants of A.
  - **IS** can request shared and intent-shared locks on all children (immediate descendants) of A.
  - **IX** can request any lock on all children of A.
  - **SIX** can do anything that having S or IX lets it do. We may consider requesting S or IS locks on children of A redundant and prevent this possibility.

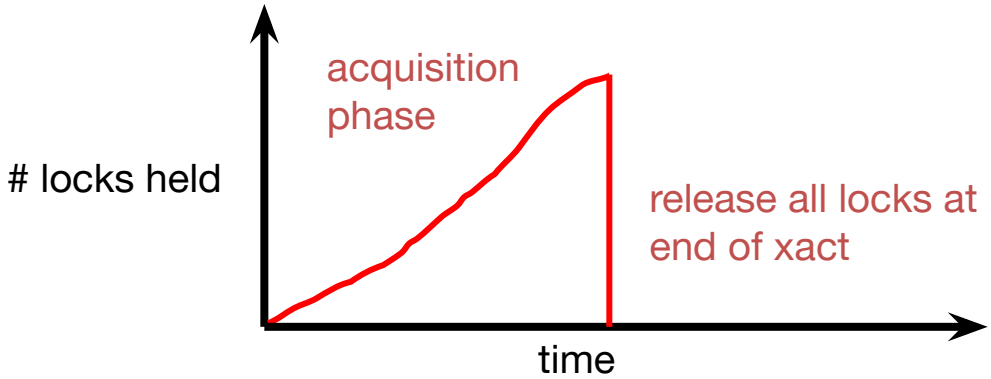
2-Phase Locking (2PL)

- One way to enforce conflict serializability
- In **2-phase locking**,
  - a transaction may not acquire a lock after it has released any lock
  - two “phases”
    - from start to until a lock is released, the transaction is just acquiring locks
    - then until the end of the transaction, it is just releasing locks



Strict 2-Phase Locking (Strict 2PL)

- The problem is that 2PL lets another transaction read new values before the transaction commits (since locks can be released long before commit)
- **Strict 2PL** avoids cascading aborts (and guarantees conflict serializability and recoverability)
  - Same as 2PL, except only allow releasing locks at end of transaction



Deadlock Detection

- We draw out a **“waits-for” graph**
  - One node for each transaction
  - If T_j *holds* a lock that conflicts with the lock that T_i wants, we add an edge from T_i to T_j
  - **A cycle indicates a deadlock** (between the transactions in the cycle) - we can abort one to end the deadlock

Deadlock Avoidance

- Typically assign priority based on start time (starting earlier means higher priority), but can use other methods (will specify on exams)
- Two approaches
  - **wait-die**: if a transaction T_i wants lock but T_j has conflicting lock
    - if T_i is higher priority, it waits for T_j to release conflicting lock
    - if T_i is lower priority, it aborts
    - transactions can only wait on lower priority transactions ➔ cannot have deadlock (lowest priority transactions cannot wait)
  - **wound-wait**: if a transaction T_i wants lock but T_j has conflicting lock
    - if T_i is higher priority, it causes T_j to abort (“wound”)
    - if T_i is lower priority, it waits for T_j to finish
    - transactions can only wait on higher priority transactions ➔ cannot have deadlock (highest priority transactions can’t wait)

Multi-granularity Locking

- Our new compatibility matrix

	NL	IS	IX	SIX	S	X
NL	✓	✓	✓	✓	✓	✓
IS	✓	✓	✓	✓	✓	✗
IX	✓	✓	✓	✗	✗	✗
SIX	✓	✓	✗	✗	✗	✗
S	✓	✓	✗	✗	✓	✗
X	✓	✗	✗	✗	✗	✗

Recovery

Recovery Policies

- **Steal/No Force**
  - **Steal** - Uncommitted transactions can overwrite the most recent committed value of an object on disk
    - Necessitates UNDO for Atomicity (all or none of Xact's operations persist)
  - **No Force -** *Don't have to* write all pages modified by a transaction from the buffer cache to disk before committing the transaction
    - Necessitates REDO for Durability (not losing results of committed Xacts)
  - Harder to enforce atomicity and durability, but gives best performance
- **No Steal** locks buffer pages from optimal use, but keeps uncommitted changes away from disk (easy atomicity)
- **Force** necessitates extra writes on commit, but everything is guaranteed to be there (easy durability)

Write-Ahead Logging

1. **Log records must be on disk before the data page gets written to disk.**
    - How we achieve atomicity
    - Can’t undo an operation if data page written before log - don’t know operation happened
2. **All log records must be written to disk when a transaction commits.**
    - How we achieve durability
    - We know what operations to redo in case of crash

Undo Logging

- Write log records to ensure **atomicity** after a system crash:
  - **<START T>**: transaction T has begun
  - **<COMMIT T>**: T has committed
  - **<ABORT T>**: T has aborted
  - **<T,X,v>**: T has updated element X, and its **old** value was v
- If T commits, then **FLUSH(X)** must be written to disk before **<COMMIT T>**
  - **Force** – we can UNDO any modifications if a Xact crashes before COMMIT
- If T modifies X, then **<T,X,v>** log entry must be written to disk before **FLUSH(X)**
  - **Steal** – we can UNDO any modifications if a Xact crashes before FLUSH

Recovery with Undo Log

- Determine which txns are completed
- Undo all modifications made by incomplete txns
- Recovery manager must:
  - Read log from the **end**:
    - Actions to perform on records:
      - <T, X, v>: if T is not completed
        then write X=v to disk
        else ignore (T would have committed or aborted)

- How far back in the log should we go?
  - All the way to the start (there could be a very long transaction)

Undo Log Example

Operation	t	Mem A	Mem B	Disk A	Disk B	UNDO Log
						<START T>
READ(A, t)	8	8		8	8	
t := t*2	16	8		8	8	
WRITE(A, t)	16	16		8	8	<T,A,8>
READ(B, t)	8	16	8	8	8	
t := t*2	16	16	8	8	8	
WRITE(B, t)	16	16	16	8	8	<T,B,8>
FLUSH(A)	16	16	16	16	8	
FLUSH(B)	16	16	16	16	16	
COMMIT						<COMMIT T>

Redo Logging

- Write log records to ensure **durability** after a system crash:
  - **<START T>**: transaction T has begun
  - **<COMMIT T>**: T has committed
  - **<ABORT T>**: T has aborted
  - <T,X,v>: T has updated element X, and its **new** value was v
- If T modifies X, then both **<T,X,v>** and **<COMMIT T>** must be written to disk before **FLUSH(X)**
  - **No-Steal, No-Force** – we can REDO any modifications if a Xact crashes before FLUSH

Recovery with Redo Log

- Determine which txns are completed
- Redo all updates of committed transactions
- Recovery manager must:
  - Read log from the **beginning**:
    - Actions to perform on records:
      - <T, X, v>: if T is committed
        then write X=v to disk
        else ignore (T either aborted or was not committed)

Redo Log Example

Operation	t	Mem A	Mem B	Disk A	Disk B	REDO Log
						<START T>
READ(A, t)	8	8		8	8	
t := t*2	16	8		8	8	
WRITE(A, t)	16	16		8	8	<T,A,16>
READ(B, t)	8	16	8	8	8	
t := t*2	16	16	8	8	8	
WRITE(B, t)	16	16	16	8	8	<T,B,16>
COMMIT						<COMMIT T>
FLUSH(A)	16	16	16	16	8	
FLUSH(B)	16	16	16	16	16	

Undo/Redo Logging Summary

- Undo logging:
  - Uses Steal/Force policies
  - Undoes all updates for **running** transactions
- Redo logging:
  - Uses No Steal/No Force policies
  - Redoes all updates for **committed** transactions

Aries Recovery - LSNs

- **LSN (Log Sequence Number)**: stored in each log record. Unique, increasing, ordered identifier for each log record
- **flushedLSN**: stored in memory, keeps track of the most recent log record written to disk
- **pageLSN**: LSN of the last operation to update the page (in memory page may have a different pageLSN than the on disk page)
- **prevLSN**: stored in each log record, the LSN of the previous record written by the current record's transaction
- **lastLSN**: stored in the Xact Table, the LSN of the most recent log record written by the transaction
- **recLSN**: stored in the DPT, the log record that first dirtied the page since the last checkpoint
- **undoNextLSN**: stored in CLR records, the LSN of the next operation we need to undo for the current record's transaction

Recovery Structures

- **Transaction Table** - stores information on active transactions. Fields include
  - Xid (Transaction ID)
  - Status (Running, Committing, Aborting)
  - lastLSN
- **Dirty Page Table (DPT)** - tracks dirty pages (pages whose changes have not been flushed to disk)
  - pageID
  - recLSN

Record Types

- Records have LSN, common fields include xid (transaction ID), pageID (for modified page), type
- **UPDATE** - write operation (SQL insert/update/delete). Also includes fields for offset (where data change started), length (how much data was changed), old_data (old version of changed data - used for undos), new_data (updated version of data - used for redos)
- **COMMIT** - Xact is beginning committing process (ARIES: flush log up to and including COMMIT record)
- **ABORT** - Xact is beginning aborting process (ARIES: begin writing CLRs for undone UPDATEs)
    - **Compensation Log Record (CLR)** - indicates a given UPDATE has been undone
- **END** - Xact is finished (as in, finished committing or aborting)

Record Types (cont.)

- Checkpoint Records
  - Useful for ARIES analysis so we don’t start from very beginning of log
  - Checkpoint serves as snapshot of Xact Table/DPT
  - Fuzzy checkpoints - Xacts operating during checkpoint; Xact
  - **BEGIN CHECKPOINT** - checkpoint start, earliest point Xact Table/DPT could represent
  - **END CHECKPOINT** - checkpoint end, holds Xact Table/DPT snapshot
- **Master Record** - stores location of most recent checkpoint for recovery purposes, usually LSN 0

ATT: Xact Table
- lastLSN
- status

Dirty Page Table
- recLSN

flushedLSN

Buffer pool

Log tail

LogRecords
- LSN
- prevLSN
- XID
- type
- pageID
- length
- offset
- before-image
- after-image

Data pages
- each
- with a
- pageLSN

Master record

ARIES: Overview

ARIES: Analysis (Part 1)

- Reconstructing Xact Table and DPT
- Need to know which transactions started/committed/aborted, which pages dirtied
- Start from **begin** checkpoint log record (or start of log), go until end of log
- On any record that is not an END record:
  - Add the Xact to the Xact Table if not in table
  - Set the lastLSN of the transaction to the current operation's LSN
  - If the record is a COMMIT or an ABORT record, change the status of Xact to Committing/Aborting
- If the record is an UPDATE record and the page being updated is not in the DPT, add the page to the DPT and set recLSN equal to the LSN
- If the record is an END record, remove the transaction from the Xact table.

ARIES: Analysis (Part 2)

- After going through the log, clean up the Xact Table
- For each Xact in the Xact Table:
  - Write END records for committing Xacts. Because they’re committing, they must be finished - preserve durability
  - For running Xacts, change status to aborting and write ABORT record - preserve atomicity since not finished

ARIES: Redo

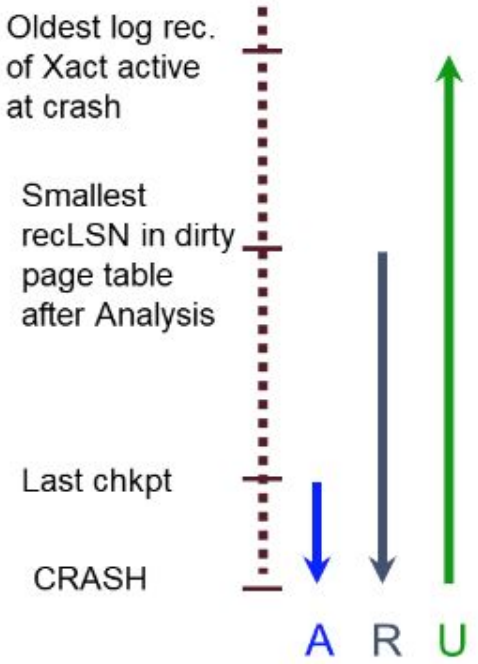
- Redo *updates and CLRs* from the earliest recLSN in the DPT to get back unflushed changes from before crash, unless:
  - page not in DPT
    - page on disk must be up to date, since we have no changes!
  - recLSN of page > LSN
    - no need to undo here: recLSN of page is *first* record that dirtied page, so this change must have been flushed
  - pageLSN (disk) >= LSN
    - page LSN for disk (LSN of last record with change written to disk) is the authoritative source for determining which changes have been applied in disk
  - Redo with after-image (redo state), update pageLSNs as you go

ARIES: Undo

- Undo each Xact in the Xact Table
  - Only UNDO updates (ignore CLRs)
- Start at end of log and work backwards to the beginning
- For every UPDATE the undo phase undoes, write a corresponding CLR to the log.
  - undoNextLSN stores the LSN of the next operation to be undone for that transaction (the prevLSN of the operation that you are undoing).
- Once you have undone all the operations for a transaction, write the END record for that transaction to the log.

ARIES: Overall

- Why does redo happen before undo?
  - If failure happens during redo or undo, next recovery can pick up what previous recovery has left and continue
    - E.g. Crash while writing CLRs in UNDO, we have to redo them
- When are transactions removed from the xact table?
  - END log record
- When is a page removed from the DPT?
  - When that page flushed to disk (pages aren't necessarily flushed to disk on commit - no force)



Worksheet

Transactions & Concurrency 1

Question 1

Is Transaction 1 doing Two-Phase Locking so far?

Answer: No; we have a lock (6) after unlock (5).

```
Txn 1:  IX-Lock(Database)    (1)
Txn 1:  IX-Lock(Table A)     (2)
Txn 1:  X-Lock(Row A1)       (3)
Txn 1:  Write(Row A1)        (4)
Txn 1:  Unlock(Row A1)       (5)
Txn 1:  S-Lock(Row A2)       (6)
Txn 2:  IX-Lock(Database)    (7)
Txn 2:  IX-Lock(Table A)     (8)
Txn 2:  X-Lock(Row A1)       (9)
Txn 2:  Write(Row A1)        (10)
Txn 2:  S-Lock(Row A2)       (11)
Txn 2:  Read(Row A2)         (12)
```

Question 2

Is Transaction 1 doing Strict Two-Phase Locking?

Answer: No; it's not even Two-Phase.

```
Txn 1:  IX-Lock(Database)      (1)
Txn 1:  IX-Lock(Table A)       (2)
Txn 1:  X-Lock(Row A1)         (3)
Txn 1:  Write(Row A1)          (4)
Txn 1:  Unlock(Row A1)         (5)
Txn 1:  S-Lock(Row A2)         (6)
Txn 2:  IX-Lock(Database)      (7)
Txn 2:  IX-Lock(Table A)       (8)
Txn 2:  X-Lock(Row A1)         (9)
Txn 2:  Write(Row A1)          (10)
Txn 2:  S-Lock(Row A2)         (11)
Txn 2:  Read(Row A2)           (12)
```

Question 3

Is this schedule conflict-serializable so far? If not, what is the cycle?

Answer: Yes, it is conflict-serializable; there is only one conflict (4 -> 10), so it is serializable to a serial order of Txn 1 -> Txn 2.

```
Txn 1:  IX-Lock(Database)    (1)
Txn 1:  IX-Lock(Table A)     (2)
Txn 1:  X-Lock(Row A1)       (3)
Txn 1:  Write(Row A1)        (4)
Txn 1:  Unlock(Row A1)       (5)
Txn 1:  S-Lock(Row A2)       (6)
Txn 2:  IX-Lock(Database)    (7)
Txn 2:  IX-Lock(Table A)     (8)
Txn 2:  X-Lock(Row A1)       (9)
Txn 2:  Write(Row A1)        (10)
Txn 2:  S-Lock(Row A2)       (11)
Txn 2:  Read(Row A2)         (12)
```

Question 4

Is this schedule serial so far?

Answer: Yes! Since all of Txn 1's operations are before Txn 2's operations, it is actually already a serial order.

```
Txn 1:  IX-Lock(Database)   (1)
Txn 1:  IX-Lock(Table A)    (2)
Txn 1:  X-Lock(Row A1)      (3)
Txn 1:  Write(Row A1)       (4)
Txn 1:  Unlock(Row A1)      (5)
Txn 1:  S-Lock(Row A2)      (6)
Txn 2:  IX-Lock(Database)   (7)
Txn 2:  IX-Lock(Table A)    (8)
Txn 2:  X-Lock(Row A1)      (9)
Txn 2:  Write(Row A1)       (10)
Txn 2:  S-Lock(Row A2)      (11)
Txn 2:  Read(Row A2)        (12)
```

Question 5

Is Transaction 2 doing Two-Phase Locking so far?

Answer: Yes. All locks (< 17) are before unlocks (17, 18).

Continuing with the operations...

```
Txn 2:  SIX-Lock(Table B)      (13)
Txn 2:  X-Lock(Row B1)         (14)
Txn 2:  Write(Row B1)          (15)
Txn 2:  Unlock(Row B1)         (16)
Txn 2:  Unlock(Table B)        (17)
Txn 1:  SIX-Lock(Table B)      (18)
Txn 1:  X-Lock(Row B1)         (19)
Txn 1:  Read(Row B1)           (20)
```

Question 6

Is Transaction 2 doing Strict Two-Phase Locking so far?

Answer: No! For strict two-phase, we must **commit** the transaction **before doing any unlocks**.

Continuing with the operations...

```
Txn 2:  SIX-Lock(Table B)      (13)
Txn 2:  X-Lock(Row B1)         (14)
Txn 2:  Write(Row B1)          (15)
Txn 2:  Unlock(Row B1)         (16)
Txn 2:  Unlock(Table B)        (17)
Txn 1:  SIX-Lock(Table B)      (18)
Txn 1:  X-Lock(Row B1)         (19)
Txn 1:  Read(Row B1)           (20)
```

Question 7

Is this schedule conflict-serializable so far? If not, what is the cycle?

Answer: No, not anymore. We have T1 -> T2 from conflict 4 -> 10, and T2 -> T1 from conflict 15 -> 20. The cycle is T1 <-> T2.

Continuing with the operations...

```
Txn 2:  SIX-Lock(Table B)      (13)
Txn 2:  X-Lock(Row B1)         (14)
Txn 2:  Write(Row B1)          (15)
Txn 2:  Unlock(Row B1)         (16)
Txn 2:  Unlock(Table B)        (17)
Txn 1:  SIX-Lock(Table B)      (18)
Txn 1:  X-Lock(Row B1)         (19)
Txn 1:  Read(Row B1)           (20)
```

Question 8

Suppose we start a new transaction, Transaction 3. What kind of locks can Transaction 3 acquire on the whole database?

The database currently has two IX locks held by Txn 1 and Txn 2. Looking at the compatibility matrix, we see that **IS and IX** are the locks compatible with IX locks, so these are the locks that Txn 3 can acquire on the database.

Continuing with the operations...

```
Txn 2:  SIX-Lock(Table B)      (13)
Txn 2:  X-Lock(Row B1)         (14)
Txn 2:  Write(Row B1)          (15)
Txn 2:  Unlock(Row B1)         (16)
Txn 2:  Unlock(Table B)        (17)
Txn 1:  SIX-Lock(Table B)      (18)
Txn 1:  X-Lock(Row B1)         (19)
Txn 1:  Read(Row B1)           (20)
```

Question 9

Given the above answers, what kind of locks can Transaction 3 acquire on Table A?

On the parent of Table A (the whole database), we know we can acquire IS and IX locks (from the previous question). These locks allow us to acquire S, X, IS, IX, and SIX locks below it.

Which of these can we actually acquire? Table A currently has two IX locks held by Txn 1 and Txn 2, so the compatibility matrix says **IS and IX** locks only.

Continuing with the operations...

```
Txn 2:  SIX-Lock(Table B)   (13)
Txn 2:  X-Lock(Row B1)      (14)
Txn 2:  Write(Row B1)       (15)
Txn 2:  Unlock(Row B1)      (16)
Txn 2:  Unlock(Table B)     (17)
Txn 1:  SIX-Lock(Table B)   (18)
Txn 1:  X-Lock(Row B1)      (19)
Txn 1:  Read(Row B1)        (20)
```

Question 10

Given the above answers, what kind of locks can Transaction 3 acquire on Row A3?

On the parent of Row A3 (Table A), we know we can acquire IS and IX locks (from the previous question). These locks allow us to acquire S, X, IS, IX, and SIX locks below it.

Which of these can we actually acquire? Row A3 has no locks currently, so we can acquire any of these! But this is a leaf node and I locks are for intermediate nodes only, so in practice we can only acquire **S and X locks**.

Continuing with the operations...

```
Txn 2:  SIX-Lock(Table B)      (13)
Txn 2:  X-Lock(Row B1)         (14)
Txn 2:  Write(Row B1)          (15)
Txn 2:  Unlock(Row B1)         (16)
Txn 2:  Unlock(Table B)        (17)
Txn 1:  SIX-Lock(Table B)      (18)
Txn 1:  X-Lock(Row B1)         (19)
Txn 1:  Read(Row B1)           (20)
```

Question 11

Given the above answers, what kind of locks can Transaction 3 acquire on Table B?

On the parent of Table B (the whole database), we know we can acquire IS and IX locks (from the previous question). These locks allow us to acquire S, X, IS, IX, and SIX locks below it.

Which of these can we actually acquire? Table B currently has an SIX lock, so the compatibility matrix says **IS locks** only.

Continuing with the operations...

```
Txn 2:  SIX-Lock(Table B)   (13)
Txn 2:  X-Lock(Row B1)      (14)
Txn 2:  Write(Row B1)       (15)
Txn 2:  Unlock(Row B1)      (16)
Txn 2:  Unlock(Table B)     (17)
Txn 1:  SIX-Lock(Table B)   (18)
Txn 1:  X-Lock(Row B1)      (19)
Txn 1:  Read(Row B1)        (20)
```

Question 12

Given the above answers, what kind of locks can Transaction 3 acquire on Row B2?

On the parent of Row B2 (Table B), we know we can acquire IS locks (from the previous question). These locks allow us to acquire S and IS locks below it.

Which of these can we actually acquire? Row B2 has no locks currently, so we can acquire any of these! But this is a leaf node and I locks are for intermediate nodes only, so in practice we can only acquire the **S lock**.

Continuing with the operations...

```
Txn 2:  SIX-Lock(Table B)      (13)
Txn 2:  X-Lock(Row B1)         (14)
Txn 2:  Write(Row B1)          (15)
Txn 2:  Unlock(Row B1)         (16)
Txn 2:  Unlock(Table B)        (17)
Txn 1:  SIX-Lock(Table B)      (18)
Txn 1:  X-Lock(Row B1)         (19)
Txn 1:  Read(Row B1)           (20)
```

Question 13

What kind of locks can Transaction 1 acquire on Row B2?

Transaction 1 currently holds an SIX lock on the parent of Row B2 (Table B), which allows it to acquire X and IX locks below it.

Which of these can it actually acquire? Row B2 has no locks, so it can acquire either of them; however, I locks are for intermediate nodes only, so it can just acquire the **X lock**.

Continuing with the operations...

```
Txn 2:  SIX-Lock(Table B)      (13)
Txn 2:  X-Lock(Row B1)         (14)
Txn 2:  Write(Row B1)          (15)
Txn 2:  Unlock(Row B1)         (16)
Txn 2:  Unlock(Table B)        (17)
Txn 1:  SIX-Lock(Table B)      (18)
Txn 1:  X-Lock(Row B1)         (19)
Txn 1:  Read(Row B1)           (20)
```

Question 14

We have now entered a deadlock. What is the waits-for cycle between the transactions?

T1 (26) waits on T3 (23).

T3 (25) waits on T2 (9).

T2 (21) waits on T1 (18).

Continuing with the operations...

```
Txn 2:  IX-Lock(Table B)          (21)
Txn 3:  IX-Lock(Database)         (22)
Txn 3:  X-Lock(Table C)           (23)
Txn 3:  IX-Lock(Table A)          (24)
Txn 3:  X-Lock(Row A1)            (25)
Txn 1:  IX-Lock(Table C)          (26)
```

Question 15

We can end this deadlock by aborting the youngest transaction. Which transaction do we abort?

The youngest transaction is **T3**, so we abort that one.

Continuing with the operations...

```
Txn 2:  IX-Lock(Table B)        (21)
Txn 3:  IX-Lock(Database)       (22)
Txn 3:  X-Lock(Table C)         (23)
Txn 3:  IX-Lock(Table A)        (24)
Txn 3:  X-Lock(Row A1)          (25)
Txn 1:  IX-Lock(Table C)        (26)
```

Question 16

If we were using **wound-wait**, what is the **first operation** in this sequence that would cause a transaction to get aborted, and which transaction gets aborted?

In **wound-wait**, the only waiting that happens is lower priority waiting for higher priority. If a higher priority transaction tries to wait, it will just abort ("wound") the transaction it is waiting on.

At (21), T2 can wait on T1. At (25), T3 can wait on T2. **But at (26), T1 will not wait on T3; it will instead abort T3.**

Continuing with the operations...

```
Txn 2:  IX-Lock(Table B)      (21)
Txn 3:  IX-Lock(Database)     (22)
Txn 3:  X-Lock(Table C)       (23)
Txn 3:  IX-Lock(Table A)      (24)
Txn 3:  X-Lock(Row A1)        (25)
Txn 1:  IX-Lock(Table C)      (26)
```

Question 17

If we were using **wait-die**, what is the **first operation** in this sequence that would cause a transaction to get aborted, and which transaction gets aborted?

In **wait-die**, the only waiting that happens is higher priority waiting for lower priority. If a lower priority transaction tries to wait, it will just abort itself ("die") instead.

At (21), T2 will not wait on T1, since T2 is lower priority. Thus, T2 will abort.

Continuing with the operations...

```
Txn 2:  IX-Lock(Table B)      (21)
Txn 3:  IX-Lock(Database)     (22)
Txn 3:  X-Lock(Table C)       (23)
Txn 3:  IX-Lock(Table A)      (24)
Txn 3:  X-Lock(Row A1)        (25)
Txn 1:  IX-Lock(Table C)      (26)
```

Transactions & Concurrency 2

Question 1

Consider a database with objects X and Y and assume that there are two transactions T1 and T2. T1 first reads X and Y and then writes X and Y. T2 reads and writes X then reads and writes Y. Create a schedule for these transactions that is **not** serializable. Explain why your schedule is not serializable.

In this example, T1 reads X before T2 writes X. However, T1 writes X after T2 reads/writes it.

The schedule is thus not serializable since there is no equivalent serial schedule that would have the same result (neither T1-T2 or T2-T1).

Transaction 1	Transaction 2
Read(X)	
Read(Y)	
	Read(X)
	Write(X)
	Read(Y)
Write(X)	
Write(Y)	
	Write(Y)

Question 2

Would your schedule be allowed under strict two-phase locking? Why or why not?

No, because strict 2PL ensures serializability. Keep in mind that strict 2PL only allows releasing locks at the end of a transaction.

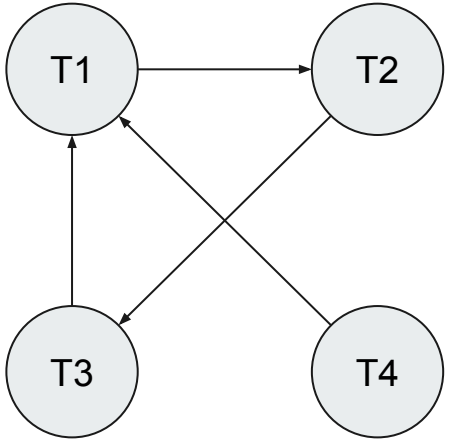
In the example schedule shown, when Transaction 2 attempts to acquire an exclusive lock to write X, it will have to wait for Transaction 1 to release its lock on X, which will not happen until Transaction 1 commits. This will never happen, so this schedule is not possible under strict 2PL.

Transaction 1	Transaction 2
Read(X)	
Read(Y)	
	Read(X)
	Write(X)
	Read(Y)
Write(X)	
Write(Y)	
	Write(Y)

Question 3

Draw the waits-for graph for this schedule.

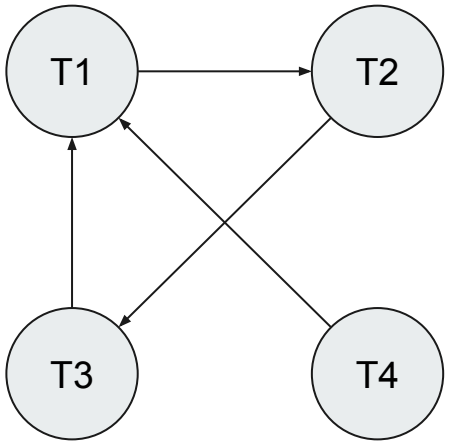
	1	2	3	4	5	6	7	8
T1				X(B)	X(A)			S(D)
T2		X(D)	S(E)					
T3	X(E)						S(B)	
T4						X(A)		



Question 4

Are there any transactions involved in deadlock?

	1	2	3	4	5	6	7	8
T1				X(B)	X(A)			S(D)
T2		X(D)	S(E)					
T3	X(E)						S(B)	
T4						X(A)		



Yes, Transactions 1, 2, & 3 are deadlocked.

Question 5

Next, assume that T1 priority > T2 > T3 > T4. You are the database administrator and one of your users purchases an exclusive plan to ensure that their transaction, Transaction 2, runs to completion. Assuming the same schedule, what deadlock avoidance policy would you choose to make sure Transaction 2 commits?

Question 5

	1	2	3	4	5	6	7	8
T1				X(B)	X(A)			S(D)
T2		X(D)	S(E)					
T3	X(E)						S(B)	
T4						X(A)		

Choose wait-die. Under wait-die, T3 and T4 abort after steps 6 and 7 because they are attempting to acquire a lock held by a transaction with higher priority. Afterwards, both T1 and T2 run to completion.

However, under wound-wait, T3 will be killed by T2 at step 3, and T2 will be killed by T1 at step 8. Since we want to make sure Transaction 2 commits, we should choose wait-die.

Recovery 1

Question 1

Consider a scenario where we update the **recLSN** in the DPT to reflect *each update* to a page, regardless of when that page was brought into the buffer pool. What bugs might you see after recovery? Select all that apply:

(a) Some writes of committed transactions would be lost
(b) Some writes of aborted transactions would be visible in the database
(c) The system tries to commit or abort a transaction that is not in the transaction table

(a) **YES**: the recLSN tells us how early to go into the log to find records to REDO. If we only track the latest LSN, then we will not go early enough to find the UPDATE records that weren't flushed
(b) **YES**: Tracking the latest LSN means we will miss CLR records written to the log before the latest LSN (these reflect UNDOs that didn't get flushed)
(c) **NO**: starting the REDO phase at a later LSN will not affect the txn table

Question 2

Suppose that you are forced to flush pages in the DPT to disk upon making a checkpoint. Which of the following cases are now guaranteed? There is one correct answer

(a) We can skip one of the three phases completely (analysis/redo/undo)
(b) We must start analysis from the beginning of the log
(c) Redo will start at the checkpoint
(d) Redo must start from the beginning of the log
(e) Undo can start at the checkpoint
(f) Undo must run until the beginning of the log

(c) Recall that REDO starts from the earliest recLSN in the DPT. If we flush all the pages in the DPT at the time of checkpoint, then we know that unflushed changes can only happen *after* the checkpoint. Thus, REDO can start from the checkpoint.

Question 3

If the buffer pool is large enough that uncommitted data are never forced to disk, is UNDO still necessary? How about REDO?

Recall that UNDO is necessary for the STEAL policy (allow an uncommitted txn to overwrite committed records on disk when it is evicted from memory). If we have enough memory in our buffer pool, we do not need to STEAL, and thus UNDOing changes is **not necessary** because all changes will still be in the buffer pool at the time of a crash.

REDO is still necessary because we will need to apply unflushed changes from before the crash

Question 4

If updates are always forced to disk when a transaction commits, is UNDO still necessary? Will ARIES perform any REDOs?

Only UNDO is necessary; we will need UNDO to finish aborting txns that were in progress at the time of a crash and weren't committing (they may have already flushed some changes to disk).

ARIES might apply some REDOs because there may be some txns in progress when the database crashes, but these will be undone in the UNDO phase

Recovery 2

Question 1

Log

LSN	Record		
10	T1	update	P1
20	T2	update	P2
30	T1	update	P2
40	T1	update	P3
50		begin-checkpoint	
60	T1	update	P4
70		end-checkpoint	
80	T1	commit	
90	T2	update	P1

Transaction Table

Txn ID	Last LSN	Txn status
T1	40	running
T2	20	running

Dirty Page Table

Page	recLSN
P1	10
P2	30

You see the **Log** table on disk after a crash, and see the **Txn table** and **DPT** when you load the checkpoint. What is the latest LSN that this checkpoint is guaranteed to be up-to-date to?

LSN 50: this is the **begin-checkpoint** record

Question 2

Log

LSN	Record		
10	T1	update	P1
20	T2	update	P2
30	T1	update	P2
40	T1	update	P3
50		begin-checkpoint	
60	T1	update	P4
70		end-checkpoint	
80	T1	commit	
90	T2	update	P1

Transaction Table

Txn ID	Last LSN	Txn status
T1	40	running
T2	20	running

Dirty Page Table

Page	recLSN
P1	10
P2	30

What do the transaction table and DPT look like at the end of analysis, and what log records do we write during analysis?

Question 2

Log

LSN	Record		
10	T1	update	P1
20	T2	update	P2
30	T1	update	P2
40	T1	update	P3
50		begin-checkpoint	
⇒60	T1	update	P4
70		end-checkpoint	
80	T1	commit	
90	T2	update	P1

Transaction Table

Txn ID	Last LSN	Txn status
T1	40	running
T2	20	running

Dirty Page Table

Page	recLSN
P1	10
P2	30
P4	60

What do the transaction table and DPT look like at the end of analysis, and what log records do we write during analysis?

Question 2

Log

LSN	Record		
10	T1	update	P1
20	T2	update	P2
30	T1	update	P2
40	T1	update	P3
50		begin-checkpoint	
60	T1	update	P4
⇒70		end-checkpoint	
80	T1	commit	
90	T2	update	P1

Transaction Table

Txn ID	Last LSN	Txn status
T1	40	running
T2	20	running

Dirty Page Table

Page	recLSN
P1	10
P2	30
P4	60

What do the transaction table and DPT look like at the end of analysis, and what log records do we write during analysis?

Question 2

Log

LSN	Record		
10	T1	update	P1
20	T2	update	P2
30	T1	update	P2
40	T1	update	P3
50		begin-checkpoint	
60	T1	update	P4
70		end-checkpoint	
⇒80	T1	commit	
90	T2	update	P1

Remember to add an **END** record because T1 committed

Transaction Table

Txn ID	Last LSN	Txn status
T1	80	committing
T2	20	running

Dirty Page Table

Page	recLSN
P1	10
P2	30
P4	60

What do the transaction table and DPT look like at the end of analysis, and what log records do we write during analysis?

Question 2

Log

LSN	Record		
10	T1	update	P1
20	T2	update	P2
30	T1	update	P2
40	T1	update	P3
50		begin-checkpoint	
60	T1	update	P4
70		end-checkpoint	
80	T1	commit	
⇒90	T2	update	P1

Remember to add an **END** record because T1 committed

Transaction Table

Txn ID	Last LSN	Txn status
T1	80	committing
T2	90	running

Remember to **ABORT** T2 because it did not commit by end of log

Dirty Page Table

Page	recLSN
P1	10
P2	30
P4	60

What do the transaction table and DPT look like at the end of analysis, and what log records do we write during analysis?

Question 2

Log

LSN	Record		
10	T1	update	P1
20	T2	update	P2
30	T1	update	P2
40	T1	update	P3
50		begin-checkpoint	
60	T1	update	P4
70		end-checkpoint	
80	T1	commit	
90	T2	update	P1
100	T1	end	

Add an **END** record because T1 committed and remove it from transaction table

Transaction Table

Txn ID	Last LSN	Txn status
T2	90	running

Dirty Page Table

Page	recLSN
P1	10
P2	30
P4	60

Remember to **ABORT** T2 because it did not commit by end of log

What do the transaction table and DPT look like at the end of analysis, and what log records do we write during analysis?

Question 2

Log

LSN	Record		
10	T1	update	P1
20	T2	update	P2
30	T1	update	P2
40	T1	update	P3
50		begin-checkpoint	
60	T1	update	P4
70		end-checkpoint	
80	T1	commit	
90	T2	update	P1
100	T1	end	
110	T2	abort	

Transaction Table

Txn ID	Last LSN	Txn status
T2	110	aborting

Dirty Page Table

Page	recLSN
P1	10
P2	30
P4	60

What do the transaction table and DPT look like at the end of analysis, and what log records do we write during analysis?

Add an **ABORT** record because T2 did not finish committing

Question 3

Log

LSN	Record		
10	T1	update	P1
20	T2	update	P2
30	T1	update	P2
40	T1	update	P3
50		begin-checkpoint	
60	T1	update	P4
70		end-checkpoint	
80	T1	commit	
90	T2	update	P1
100	T1	end	
110	T2	abort	

Transaction Table

Txn ID	Last LSN	Txn status
T2	110	aborting

Dirty Page Table

Page	recLSN
P1	10
P2	30
P4	60

The next phase of ARIES is REDO. What LSN do we start REDO from?

Start REDO from the oldest LSN in the DPT: **LSN 10**

Question 4

Log

LSN	Record		
10	T1	update	P1
20	T2	update	P2
30	T1	update	P2
40	T1	update	P3
50		begin-checkpoint	
60	T1	update	P4
70		end-checkpoint	
80	T1	commit	
90	T2	update	P1
100	T1	end	
110	T2	abort	

Transaction Table

Txn ID	Last LSN	Txn status
T2	110	aborting

Dirty Page Table

Page	recLSN
P1	10
P2	30
P4	60

From that record, we will redo the effects of all the following records, except we will not redo certain records. What are the LSNs of the records we do NOT redo?

20: the recLSN of P2 is higher than 20, so we know that LSN 20 is flushed
40: P3 is not in the DPT so it is flushed
50, 70, 80 (100, 110): these are not update operations

Question 5

Log

LSN	Record		
10	T1	update	P1
20	T2	update	P2
30	T1	update	P2
40	T1	update	P3
50		begin-checkpoint	
60	T1	update	P4
70		end-checkpoint	
80	T1	commit	
90	T2	update	P1
100	T1	end	
110	T2	abort	
120	T2	CLR nextLSN: 20	

The last phase of ARIES is undo. What do we do for this phase? Answer this question by writing out the log records that will be recorded for each step. Stop after you write your first CLR record (make sure your CLR record specifies the nextLSN!).

During UNDO, we need to undo every UPDATE for all transactions that were active at the time of the crash. This is just T2 here, so we need to write a CLR record for the update at LSN 90, and then specify the next update to undo (which is LSN 20)

Question 6

Log

LSN	Record
10	T1 update P1
20	T2 update P2
30	T1 update P2
40	T1 update P3
50	begin-checkpoint
60	T1 update P4
70	end-checkpoint
80	T1 commit
90	T2 update P1
100	T1 end
110	T2 abort
120	T2 CLR nextLSN: 20

We just crashed during ARIES recovery! You load up the checkpoint. What does the transaction table and dirty page table look like?

The transaction table and DPT look the same because we didn't write a new checkpoint

Question 7

Log

LSN	Record
10	T1 update P1
20	T2 update P2
30	T1 update P2
40	T1 update P3
50	begin-checkpoint
60	T1 update P4
70	end-checkpoint
80	T1 commit
90	T2 update P1
100	T1 end
110	T2 abort
120	T2 CLR nextLSN: 20

Transaction Table

Txn ID	Last LSN	Txn status
T2	120	aborting

Dirty Page Table

Page	recLSN
P1	10
P2	30
P4	60

You run the analysis phase. What do the transaction table and dirty page look like at the end of analysis?

Question 8

Log

LSN	Record
10	T1 update P1
20	T2 update P2
30	T1 update P2
40	T1 update P3
50	begin-checkpoint
60	T1 update P4
70	end-checkpoint
80	T1 commit
90	T2 update P1
100	T1 end
110	T2 abort
120	T2 CLR nextLSN: 20

Transaction Table

Txn ID	Last LSN	Txn status
T2	120	aborting

Dirty Page Table

Page	recLSN
P1	10
P2	30
P4	60

You run the redo phase. In order, what are the LSNs that we redo?

10: LSN is >= recLSN for P1
30: LSN is >= recLSN for P2
60: LSN is >= recLSN for P4
90: LSN is >= recLSN for P1
120: CLR record must be applied

Note that we redo T1 even though it committed

Question 9

Log

LSN	Record		
10	T1	update	P1
20	T2	update	P2
30	T1	update	P2
40	T1	update	P3
50		begin-checkpoint	
60	T1	update	P4
70		end-checkpoint	
80	T1	commit	
90	T2	update	P1
100	T1	end	
110	T2	abort	
120	T2	CLR nextLSN: 20	
130	T2	CLR nextLSN: null	
140	T2	end	

Now we run the undo phase. What do we do? (Answer again with the log records that you have to add.)

Finish adding the CLR for the aborted transaction T2, and then write an END record to indicate that we are done