CS 186 Fall 2022 Introduction to Database Systems

FINAL

INSTRUCTIONS

This is your exam. Complete it either at exam.cs61a.org or, if that doesn't work, by emailing course staff with your solutions before the exam deadline.

This exam is intended for the student with email address `<EMAILADDRESS>`. If this is not your email address, notify course staff immediately, as each exam is different. Do not distribute this exam PDF even after the exam ends, as some students may be taking the exam in a different time zone.

For questions with **circular bubbles**, you should select exactly *one* choice.

- ◯ You must choose either this option
- ◯ Or this one, but not both!

For questions with **square checkboxes**, you may select *multiple* choices.

- ☐ You could select this choice.
- ☐ You could select this one too!

You may start your exam now. Your exam is due at <DEADLINE> Pacific Time. Go to the next page to begin.

Preliminaries

1 CS 186 Fall 2022 Final

Unless you have special accommodations, you will have 170 minutes to complete the final. Do not share this exam until solutions are released.

1.0.1 Contents:

- The Final has 10 questions, each with multiple parts, and worth a total of 186 points.

1.0.2 Aids:

- You may use 3 pages (double-sided) of handwritten notes as well as a calculator.
- You must work individually on this exam.

1.0.3 Grading Notes:

- All I/Os must be written as integers. There is no such thing as 1.02 I/Os – that is actually 2 I/Os.
- 1 KB = 1024 bytes. We will be using powers of 2, not powers of 10.
- Unsimplified answers, like those left in log format, will receive a point penalty.

(a) What is your full name?

(b) What is your student ID number?

(c) SID of the person to your left (Put None if you are taking the exam remotely):

(d) SID of the person to your right (Put None if you are taking the exam remotely):

1. (10 points) Distributed Transactions

Alice, Bob, Mallory, and Eve are working on a final project and all need to agree on a topic. To do so, they decide to use 2PC with Eve as the coordinator.

Assume the following for the remainder of this question:

(a) Flushing a log record takes 50 ms
(b) All other operations (except sending messages and flushing log records) occur instantaneously
(c) All latencies are in milliseconds and are symmetric (if it takes 100 ms to send a message from Alice to Bob, it takes 100 ms to send a message from Bob to Alice as well)
(d) When a participant wants to ask the coordinator about the status of a transaction, it sends a STATUS INQUIRY message.

- It takes 150 ms for messages to travel between Alice and Eve
- It takes 200 ms for messages to travel between Alice and Bob
- It takes 250 ms for messages to travel between Bob and Eve
- It takes 450 ms for messages to travel between Bob and Mallory
- It takes 350 ms for messages to travel between Mallory and Eve
- It takes 300 ms for messages to travel between any other pairs of people

For all parts, assume the following events take place:

(a) 0 ms: Eve begins the 2PC protocol
(b) 900 ms: Bob temporarily goes offline (Bob crashes)
(c) 1500 ms: Bob recovers

For all questions below that ask for time, express your answer in ms.

(a) (4 points) 2PC walkthrough

For parts a.i to a.viii, assume Alice and Bob vote YES, but Mallory votes NO. Assume no optimizations are made with respect to the 2PC algorithm.

i. (0.5 pt) Between Alice, Bob, and Mallory (the three participants), who is the first to send a message?

● A. Alice
○ B. Bob
○ C. Mallory

At the start of the protocol, Eve will simultaneously send a prepare message to all three participants (requesting their vote). Alice has the smallest communication latency (150ms) so will be the first to receive Eve's request and thus respond with a vote (after logging as necessary).

ii. (0.5 pt) What is the first message that Eve receives?

○ A. ABORT
○ B. ACK
○ C. COMMIT
○ D. PREPARE
○ E. STATUS INQUIRY
● F. VOTE YES
○ G. VOTE NO

Eve will receive Alice's vote first as Alice is the first to respond to the request for votes, as noted in the previous solution. Alice votes yes, as stated in the prompt.

iii. (0.5 pt) What is the earliest time at which Eve has received all votes? Write **only** a single number.

750

350 + 50 + 350 = 750 ms

Mallory has the largest communication delay with Eve, so consider just this case. Eve sends a prepare message at t=0; the message takes 350 ms to travel to Mallory. Since we are **not** using presumed abort, Mallory must log and flush an abort record (50 ms) before sending a response to Eve (350 ms).

iv. (0.5 pt) At what time (if any) does Mallory send an Ack? If she never sends an Ack, write -1. Write **only** a single number.

1200

750 + 50 + 350 + 50 = 1200 ms

Given Eve receives all votes at t=750, Eve must then log and flush an abort record (50 ms) since we are not using presumed abort. Even then sends abort messages to all participants. The message takes 350 ms to reach Mallory. Mallory can then log and flush a phase 2 abort record (50 ms) before acking.

1150 was also accepted to account for the case where Mallory does not log a duplicate abort record in phase 2.

v. (0.5 pt) At what time (if any) does Eve receive her first Ack? If she never receives an Ack, write -1. Write **only** a single number.

1150

750 + 50 + 150 + 50 + 150 = 1150 ms

Eve and Alice share the lowest communication delay, so the first Ack that Eve receives will be from Alice. Given Eve receives all votes at t=750, Eve must then log and flush an abort record (50 ms) since we are not using presumed abort. Even then sends abort messages to all participants. The message takes 150 ms to reach Alice, who must then log and flush an abort record (50 ms), before sending an Ack, which takes an additional 150 ms to reach Eve.

vi. (0.5 pt) What is the earliest time at which Eve can log an End record? Write **only** a single number.

2300

1500 + 250 + 250 + 50 + 250 = 2300 ms

Eve's initial abort message to Bob is never received since Bob crashes at 900 ms. When Bob recovers at t=1500, he will see a prepare record in his log and send a status inquiry to Eve (250 ms). Eve will respond that the transaction aborted (250 ms), so Bob will log and flush an abort record (50 ms) and then send an Ack to Eve (250 ms). Only at this time, once Acks have been received from all participants, will Eve end the transaction.

vii. **(0.5 pt)** How many log records is Alice required to flush over the course of the entire protocol?

2

Alice must flush a prepare record in phase 1 and an abort record in phase 2.

viii. **(0.5 pt)** What message(s) will Bob **send** over the course of the entire protocol?

- ☐ A. ABORT
- ■ B. ACK
- ☐ C. COMMIT
- ☐ D. PREPARE
- ■ E. STATUS INQUIRY
- ■ F. VOTE YES
- ☐ G. VOTE NO

Bob will send a yes vote in phase 1 of the algorithm. As mentioned in the solution for part a.vi, Bob will also send a status inquiry and an Ack after recovering.

(b) (3 points) 2PC Walkthrough but with Presumed Abort

For parts b.i to b.vi, assume Alice and Bob still vote YES, and Mallory still votes NO. However, assume we are now using **presumed abort**.

Assume the same events as stated in the beginning of the question take place, repeated here for easy reference:

i. 0 ms: Eve begins the 2PC protocol
ii. 900 ms: Bob temporarily goes offline (Bob crashes)
iii. 1500 ms: Bob recovers

i. (0.5 pt) What is the earliest time at which Eve can now decide to abort? Even if she flushes this log record, do **not** include the time to flush. Write **only** a single number.

700

350 + 350 = 700 ms

Mallory has the largest communication delay with Eve, so consider just this case. Eve sends a prepare message at t=0; the message takes 350 ms to travel to Mallory. Since we are now using presumed abort, Mallory will log but **not** flush an abort record (0 ms) before sending a response to Eve (350 ms).

ii. (0.5 pt) At what time (if any) does Alice now send an Ack? If she never sends an Ack, write -1. Write **only** a single number.

-1

With the presumed abort optimization, we only send Acks if a transaction commits.

iii. (0.5 pt) How many log records is Bob now required to flush over the course of the entire protocol? Write **only** a single number.

1

Bob must still flush a prepare record in phase 1. However, he is no longer required to flush the abort record in phase 2.

iv. (0.5 pt) Which of the following message(s) will Mallory now **send** over the course of the entire protocol?

☐ A. ABORT
☐ B. ACK
☐ C. COMMIT
☐ D. PREPARE
☐ E. STATUS INQUIRY
☐ F. VOTE YES
■ G. VOTE NO

Mallory must send a no vote in phase 1 but no longer needs to send an Ack in phase 2 thanks to the presumed abort optimization.

v. (0.5 pt) How many messages will Eve now **receive** over the course of the entire protocol? Write **only** a single number.

4

Eve will receive 3 votes from the participants as well as 1 status inquiry from Bob when he recovers. Eve will no longer receive any Acks.

vi. (0.5 pt) What is the earliest point at which we can be guaranteed the outcome of our 2PC decision will be an abort?

- (●) A. When Mallory sends a NO vote
- ( ) B. When Eve has received all votes and logged an abort record
- ( ) C. When Eve flushes her abort record
- ( ) D. When Alice, Bob, and Mallory have all received an abort message from Eve

Once Mallory sends a no vote, we know that there is not a unanimous decision to commit, which, under 2PC, means the transaction will be aborted. Even if Mallory were to now crash (and forget the decision to vote no), when recovering, she would still perform an abort since there would be no records in her log.

(c) (1.5 points) Let's vote differently!

For parts c.i to c.iii **only**, now assume Alice, Bob, and Mallory all vote YES. Assume we still use presumed abort.

Assume the same events as stated in the beginning of the question take place, repeated here for easy reference:

i. 0 ms: Eve begins the 2PC protocol
ii. 900 ms: Bob temporarily goes offline (Bob crashes)
iii. 1500 ms: Bob recovers

i. (0.5 pt) What is the earliest time at which Alice can determine the final decision?

950

350 + 50 + 350 + 50 + 150 = 950 ms

Alice determines the final decision when she receives a commit message from Eve. Alice receives all votes at t=750 (350 + 50 + 350 since Mallory now needs to log and flush a prepare record). Alice will then log and flush a commit record (50 ms) before sending a commit message to everyone. This message takes 150 ms to reach Alice.

ii. (0.5 pt) What is the earliest time after which every person is guaranteed to commit?

800

Every person is guaranteed to commit only after Eve logs and flushes a commit record. This is because, at any time before this, Eve could crash, and, when recovering, would abort the transaction due to not seeing anything in her log.

iii. (0.5 pt) Which of the following messages are sent by any of the four people over the entire course of the protocol?

☐ A. ABORT
■ B. ACK
■ C. COMMIT
■ D. PREPARE
■ E. STATUS INQUIRY
■ F. VOTE YES
☐ G. VOTE NO

(d) (1.5 points) Conceptual True/False

The following questions are independent and unrelated to all of the previous parts. None of the previous assumptions hold (including the ones at the beginning of the question).

i. (0.5 pt) Assume the coordinator in 2PC crashes after flushing a commit record but before sending out commit messages and does not come back up. Can the rest of the participants still commit?

- ○ Yes
- ● No

Unfortunately, 2PC cannot handle indefinite failures. If the coordinator never recovers, no participant will ever receive commit messages (and thus will never commit).

ii. (0.5 pt) Two-phase locking cannot be used by distributed databases to guarantee serializability.

- ○ True
- ● False

By partitioning or centralizing/coordinating locks, distributed databases can still use 2PL.

iii. (0.5 pt) 2PC with presumed abort can result in fewer records being logged than the unoptimized version.

- ○ True
- ● False

While 2PC with presumed abort does not immediately have to flush some log records while it is running, the same log records as the unoptimized version still need to be logged.

2. (13 points) B+ Tree

(a) (7 points) World Cup Database

To keep up with this year's World Cup, FIFA decided to create a database to keep track of all current soccer players.

```
CREATE TABLE soccer {
    pid INTEGER PRIMARY KEY,  //player id
    name TEXT,
    country TEXT,
    goals INTEGER,            //total number of goals player made in this year's World Cup
    minutes FLOAT,            //total number of minutes player played in this year's World Cup
    coach TEXT
};
```

There are three different indices on this table.

- Index A: Alternative 2, unclustered B+ tree on `pid` with h = 3, d = 2
  - There are 110 leaf nodes.
  - The first record with `pid` > 130 occurs on the 90th leaf node (where the leftmost leaf node is the 1st leaf node).
- Index B: Alternative 2, clustered B+ tree on `country` with h = 3, d = 3
- Index C: Alternative 1 B+ tree on `coach` with h = 5, d = 2

Additional Assumptions:

- The `soccer` table has exactly 200 data pages.
- There are 77 records where `goals` >= 1
- There are 97 records where `coach` == 'Paulo Bento'
- There are 113 records where `country` == 'Poland'
- Each data page stores 10 records.
- Assume that for each index, all leaf nodes are full.

i. (1 pt) FIFA wants to identify all players who have scored a goal during the World Cup. What is the *worst case* I/O cost to execute the following query, using the best index if appropriate?

```
SELECT pid, name FROM soccer WHERE goals >= 1
```

200 I/Os

There is no index built on `goals`, so we must do a full scan which costs 200 I/Os.

ii. (2 pt) FIFA is curious about the relationship between coach Paulo Bento and his players' `pid`. If you have to use index A, what is the max IO cost that would be incurred to execute the following query?

```
SELECT name, country FROM soccer WHERE pid > 130 AND coach == 'Paulo Bento'
```

108 I/Os

Using Index A, we need to read the root and 2 inner nodes to get to the 90th leaf (3 I/Os). We need to read the 90th leaf up to and including the last leaf node, which is 21 leaves in total. In the worst case, every entry on the 90th leaf has pid > 130. So we need to read 21 * 4 = 84 data pages. Total I/O cost: 3 + 21 + 84 = 108 I/Os. If we don't use any index, we need to perform a full scan, which takes 200 I/Os. See next question for I/O cost of using Index C.

iii. **(2 pt)** For the same query as above, what is the *best case* I/O cost using any applicable index (including no index)?

15 I/Os

We need to read the root node and 4 inner nodes to get to the leaf nodes (5 I/Os). For Alt 1 trees, each leaf node is a data page, so each of them stores 10 records. In the best case, 97 records would be stored on 10 data pages. So the cost is 5 + 10 = 15 I/Os.

iv. **(1 pt)** After Poland's goalkeeper denied Messi a penalty kick, FIFA has a newfound interest in Polish soccer players. Using the appropriate index, what is the *best case* I/O cost to execute the following query?

```
SELECT * FROM soccer WHERE country == 'Poland'
```

34 I/Os

We need to read root node and 2 inner nodes (3 I/Os). In the best case, 113 entries can be stored across ceil(113/6)=19 leaves. Since the tree is clustered, we only need to read ceil(113/10)=12 data pages. So the total cost is 3 + 19 + 12 = 34 I/Os.

v. **(1 pt)** If we wanted to perform a full index scan of the table, which index incurs the least amount of I/Os?

- ○ A. Index A
- ○ B. Index B
- ○ C. Index C
- ● D. None, all indices yield the same I/O cost

Note: For this question, it was later decided that credit was given to everyone who provided an answer.

(b) (6 points) Legacy Database

The decision to construct a database for this year's World Cup was wildly successful; as a result, FIFA decides to build a more comprehensive database encompassing every World Cup in history called `legacy`. To speed up their queries, FIFA wants to build an index for `legacy` on the `name` attribute.

i. (2 pt) Suppose FIFA uses bulkloading to efficiently create a B+ tree using `name` as the key. Assuming d = 6, the fill factor is 0.5, and FIFA has 1200 records in `legacy`, how many leaf nodes are in the resulting B+ tree?

200

Each leaf can store 2d(fill_factor) entries, which is 6 in this case. It would take 1200/6=200 leaves to store 1200 records.

ii. (2 pt) What is the minimum height of a bulkloaded B+ tree with the same constraints as above, except the fill factor is increased to 0.8?

3

Each leaf can now store 9 entries. So there would be ceil(1200/9) = 134 leaves. The fanout for a bulkloaded tree is d+1=7. So there will be ceil(134/7)=20 inner nodes at the level above the leaf nodes. Above that level, there will be another ceil(20/7)=3 inner nodes, and then there will be ceil(3/7)=1 inner node (the root node) at the level above the 3 inner nodes. This will be a height 3 B+ tree.

iii. (2 pt) In one or two sentences, provide one disadvantage of increasing the fill factor for bulkloading.

Credit is given for any answer that mentions an increased fill factor leading to less room to insert values after the B+ tree is created. Therefore, if we want to insert a large number of values, we might need to change the height of the B+ tree.

3. (33 points) Recovery

(a) (2 pt) Which of the following choices are true statements? Mark all that apply.

- [x] A. Redo Logging has lower query throughput during normal execution on average compared to Undo Logging.
- [ ] B. ARIES is designed such that we skip the Redo phase if we are given a log where no transactions commit.
- [x] C. When a transaction aborts during Undo Logging, we first rollback the transaction's changes and flush before writing an abort log record to disk.
- [ ] D. Assume that a system only allows at most n active transactions. In the worst case, there is no ACID-preserving mechanism to bound the number of operations undone in the undo phase by some constant k.
- [ ] E. None of the above.

True. Redo Logging has less throughput because it utilizes a no-steal policy, which constrains the number of transactions we can have running at the same time due to limited buffer pages.

False. We do not consider transaction status when doing the redo phase.

True. Because Undo Logging utilizes a force policy, rollbacks should be flushed to disk before the abort record is written.

False. We can force the system to abort transactions that are longer than $\frac{k}{n}$ operations long. A trivial approach is to also just always abort all transactions.

(b) (8 points) Which Policy?

For the following scenarios, select the policy(s) which **must be in effect** in the given scenario. Select at least one answer.

i. (2 pt) A transaction `T1` wants to do an operation which requires a page to be brought into memory. However, all the available buffer pages are used up. The buffer manager then writes to disk and evicts a page which contains changes from a transaction `T2`, where `T2` did not commit yet.

- [ ] A. Force
- [ ] B. No Force
- [x] C. Steal
- [ ] D. No Steal

Force is not guaranteed because no information is given about the case when a transaction commits.

No force is not guaranteed because it is possible we are using a Force policy.

Steal is guaranteed because the buffer manager evicted a page containing uncommitted changes from `T2`.

No steal is not guaranteed because we evicted a page belonging to `T2`, which did not commit yet.

ii. **(2 pt)** The buffer manager is utilizing MRU as its replacement policy and all the available buffer pages are currently full. Transactions `T1` and `T2` modify the same in-memory page. A new transaction `T3` does an operation which brings a new page into memory. `T1` aborts afterwards. We later see `T1` and `T2`'s changes from earlier is flushed to disk.

☐ A. Force

■ B. No Force

■ C. Steal

☐ D. No Steal

Force is not guaranteed because we did not mention commit anywhere

No-force is guaranteed because due to the buffer manager's MRU page replacement policy, the page `T3` brought in will evict the page `T1` and `T2` modified. This is the definition of no-force

Steal is guaranteed because we see the changes from `T1` and `T2` are flushed to disk, even though none of the transactions committed.

No-steal is not guaranteed because we evidence of steal being the chosen policy.

iii. **(2 pt)** Let P be the set of all in-memory pages. Let $MP_T \subseteq P$ be the set of in-memory pages a transaction T modifies. Assume T commits at timestamp t. We adopt a logging scheme such that:

flushedLSN $\geq$ on-disk pageLSN$_p$ $\forall p \in P$

At timestamp t, $\forall p \in MP_T$, $p \notin$ DPT

At timestamp t, LSN of T's commit record $>$ on-disk pageLSN$_p$ $\forall p \in MP_T$

■ A. Force

☐ B. No Force

■ C. Steal

☐ D. No Steal

The inequalities describe Undo Logging, which utilizes a force steal policy.

iv. **(2 pt)** Several transactions are running on our database. While monitoring the database performance, you notice you are limited to only having a few uncommitted transactions running concurrently due to memory constraints. You do not see high amounts of I/O being incurred when a transaction commits. You see pages are being evicted and flushed in an efficient manner by the buffer manager.

☐ A. Force

■ B. No Force

☐ C. Steal

■ D. No Steal

Limited to a few uncommitted transactions due to memory constraints implies the no-steal policy, as pages cannot be evicted from memory until the transaction commits. Not seeing high amounts of I/O being incurred when a transaction commits pimples we are not using the force policy. Seeing pages being evicted and flushed in an efficient manager by the buffer manager implies no force.

(c) (12 points) Implementing Redo Logging

Let's implement Redo Logging! Some useful project code information has been copied below for your convenience:

```
public class TransactionTableEntry {
    // lastLSN of transaction
    long lastLSN;

    ... // other fields
}

public abstract class LogRecord {
    /**
     * Gets the transaction number of a log record, if applicable
     * @return optional instance containing transaction number
     */
    public Optional<Long> getTransNum(){ ... }

    /**
     * @return type of log entry
     */
    public LogType getType(){ ... }
}

public class LogManager implements ... {
    /**
     * Flushes the log to at least the specified record,
     * essentially flushing up to and including the page
     * that contains the record specified by the LSN.
     * @param LSN LSN up to which the log should be flushed
     */
    public synchronized void flushToLSN(long LSN) { ... }
}
```

i. (2 points) Commit

We will first implement `commit` for RedoLogging's forward processing.

```
public RedoLoggingRecoveryManager {
    /* The set of transaction numbers of all transactions which have committed */
    Set<Long> committedTransactions;
    Map<Long, TransactionTableEntry> transactionTable;
    LogManager logManager;

    ... // other methods

    public long commit(long transNum) {
        TransactionTableEntry tableEntry = this.transactionTable.get(transNum);
        CommitTransactionLogRecord logRecord = new CommitTransactionLogRecord(
            transNum,
            tableEntry.lastLSN
        );
        long lsn = logManager.appendToLog(logRecord);
        __(A)__ = lsn;
        __(B)__;
        return lsn;
    }
```

```
    ... // other methods
}
```

A. (1 pt) Fill in the code that should replace (A).

```
tableEntry.lastLSN
```

B. (1 pt) Fill in the code that should replace (B).

```
committedTransactions.add(transNum) or logManager.flushToLSN(lsn)
```

There is a small mistake in the exam question. We meant to include `logManager.flushToLSN(lsn)` after blank A and before blank B. As a result, credit was given to both answers.

ii. (7 points) Recovery

Let's implement recovery for redo logging! Note that this is different from ARIES. Assume we only have the following log types: `LogType.UPDATE_PAGE`, `LogType.END_TRANSACTION`, `LogType.COMMIT_TRANSACTION`, `LogType.ABORT_TRANSACTION`.

HINT: Think about what policies Redo Logging uses and how this affects what we need to do to ensure durability.

```
public RedoLoggingRecoveryManager {
    /* The set of transaction numbers of all transactions which have committed */
    Set<Long> committedTransactions;
    Map<Long, TransactionTableEntry> transactionTable;
    LogManager logManager;

   ... // other methods

    public void recovery() {
        MasterLogRecord masterRecord = (MasterLogRecord) record;
        Iterator<LogRecord> iter = this.logManager.scanFrom(masterRecord.LSN);
        while (iter.hasNext()) {
            LogRecord logRecord = iter.next();
            LogType logType = logRecord.getType();
            Optional<Long> transNum = logRecord.getTransNum();
            if (__(A)__) {
                committedTransactions.add(transNum.get());
            }
        }

        Iterator<LogRecord> iter = this.logManager.scanFrom(__(B)__);
        while (iter.hasNext()) {
            LogRecord logRecord = iter.next();
            LogType logType = logRecord.getType();
            Optional<Long> transNum = logRecord.getTransNum();
            if (__(C)__) {
                continue;
            }
            if (__(D)__) {
                logRecord.redo(...)
            }

            /* rest of implementation hidden */

        }
    }

    ... // other methods
}
```

A. (2 pt) Fill in the code that should replace (A).

```
logType == LogType.COMMIT_TRANSACTION
```

B. (1 pt) Fill in the code that should replace (B).

```
masterRecord.LSN
```

C. (2 pt) Fill in the code that should replace (C).

```
!committedTransactions.contains(transNum)
```

D. (2 pt) Fill in the code that should replace (D).

```
logType == LogType.UPDATE_PAGE
```

iii. (3 points) Page Flush Hook

In project 5, we abstracted away the role of the buffer manager from the logging/recovery process. We will now implement the method which is called every time a page is considered for eviction by the buffer manager.

```
public RedoLoggingRecoveryManager {

    Set<Long> committedTransactions;
    Map<Long, TransactionTableEntry> transactionTable;
    LogManager logManager;

    ... // other methods

    /**
     * Returns the transaction number which modified
     * a given page, if applicable.
     *
     * Assume each page can only be modified by one transaction.
     * A different transaction can only modify the page when
     * the original transaction commits and the page is flushed.
     *
     * @return optional instance containing transaction number
     */
    public Optional<Long> getModifyingTransaction(...) {
        /* implementation hidden */
    }

    /**
     * Called if a page is being considered for
     * eviction by the buffer manager. This
     * method is never called on a log page.
     *
     * The method should flush the log as far as necessary.
     */
    public void pageFlushHook(...) {
        Optional<Long> transNum = getModifyingTransaction(...);

        // The case where no transaction modifies this page
        // You do not have to worry about this case
        if (transNum.isEmpty()) {
            return;
        }

        TransactionTableEntry tableEntry = this.transactionTable.get(transNum.get());
        if (__(A)__) {
            throw new Exception(``cannot evict page'');
        }
        logManager.flushToLSN(__(B)__);
    }

    ... // other methods
}
```

A. (2 pt) Fill in the code that should replace (A).

```
!committedTransactions.contains(transNum)
```

B. (1 pt) Fill in the code that should replace (B).

```
tableEntry.lastLSN
```

(d) (11 points) ARIES

We are given the following log when our database recovers from a crash:

	LSN	Description	prevLSN
0		MASTER: checkpoint at LSN 80	-
10		T1 Updates P1	-
20-70		[LOGS]	-
80		Begin Checkpoint	-
90		T2 Updates P2	70
100		T3 Updates P1	-
110		T1 COMMITS	40
120		T2 aborts	90
130		CLR undo T2 LSN: 90	120
140		End Checkpoint	-
150		T3 updates P3	100
160		T3 updates P4	150
170		T4 updates P1	-
180		T5 updates P2	-
190		T3 updates P3	160

Transaction Table:

Transaction	Status	lastLSN
T1	Running	40
T2	Running	70

DPT:

PageID	recLSN
P2	50
P4	20

The first step in performing recovery is to run the Analysis Phase. Answer the following questions about this phase.

i. (2 pt) What pages are in the DPT at the end of the analysis phase, and what are their LSNs?

- [x] A. P1 - 100
- [ ] B. P1 - 170
- [x] C. P2 - 50
- [ ] D. P2 - 90
- [x] E. P3 - 150
- [x] F. P4 - 20
- [ ] G. P4 - 160
- [ ] H. None of the above.

ii. **(1 pt)** How many transactions have their status as aborting at the end of the analysis phase, if any? Write your answer as an integer value.

4 (T2, T3, T4, T5).

iii. **(3 pt)** Sabrina wants to benchmark the three different types of buffer replacement policies we learned in class to see which one is most optimal when doing Analysis on the log above (optimal means highest hit rate). Assume we have 3 buffer pages total which are initially empty.

What is the access pattern of the data pages recorded in the log? Express your answer using capital letters followed by numbers for the pages, and separate each access with a comma. For example, if you believe that according to the log, P1, P2, and P3 are accessed in that order, then you should write P1, P2, P3.

P2, P1, P2, P3, P4, P1, P2, P3

Due to confusion in the problem wording, P1, P2, P1, P2, P3, P4, P1, P2, P3 was also given full credit.

iv. **(2 pt)** Sabrina believes MRU is the most optimal replacement policy out of the ones we learned in class for the access pattern above. Do you agree with this? Respond with yes or no. If no, please explain your answer using **1 sentence**.

Yes.

v. (3 pt) Assume Sabrina decides to use MRU as her buffer replacement policy. After running analysis on the log above, what will the `flushedLSN` be? Assume the log is only flushed as far as necessary. Your answer should be a LSN value.

180

When running MRU, we see two evictions occur. The first one is when P3's update corresponding to LSN 150 is evicted for P4's update at LSN 160. Due to WAL, $\text{pageLSN}_3 \leq \text{flushedLSN}$ must hold. Because the in-memory $\text{pageLSN}_3 = 150$, we have to flush the log up to LSN 150. The second eviction occurs when P2's update corresponding to LSN 180 is evicted for P3's update at LSN 190. Similarly, due to WAL, $\text{pageLSN}_2 \leq \text{flushedLSN}$ must hold. Because the in-memory $\text{pageLSN}_2 = 180$, we have to flush the log up to LSN 180. The flushedLSN is then 180.

4. (14 points) Disks/Files/Buffers

(a) (1 pt) In the page footer of slotted page layout, it is possible to have more free entries than filled entries.

- [x] True
- [ ] False

(b) (1 pt) If we are using bitmaps in record headers, we always need to allocate one bit for each variable length field.

- [ ] True
- [x] False

(c) (12 points)

For the rest of the question, consider the following schema and assumptions:

Schema:

```
CREATE TABLE Student (
    sid INTEGER PRIMARY KEY,
    name VARCHAR(30) NOT NULL,
    birthyear INTEGER,
    units_taken INTEGER NOT NULL,
    graduated boolean NOT NULL
)
```

Assumptions:

- Integers are 4 bytes and pointers are 4 bytes.
- Record headers contain a bitmap. Bitmaps should be as small as possible, rounded up to the nearest byte.
- Each data page is 1 KB (1 KB = 1024 B).
- The Student table is stored as a heap file using the page directory implementation.
  - There are 15 header pages, each with 20 entries. All entries are used.
- The Student table has 5000 records in total and contains students born from 1996-2005, each `birthyear` having the equal amount of students.
- If a question requires the use of slotted pages, the implementation of slotted pages will be the same as seen in lecture, discussion, and course notes.

i. (1 pt) What is the maximum size (in bytes) of a record from the Student table?

48 B

1 (bitmap) + 4 (sid) + 4 (ptr for name) + 30 (name) + 4 (birthyear) + 4 (units_taken) + 1 (graduated) = 48 bytes

ii. (1 pt) What is the maximum number of Student records that can fit onto one data page?

39 records

Smallest record size is 1 (bitmap) + 4 (sid) + 4 (ptr for name) + 0 (name) + 4 (birthyear) + 4 (units_taken) + 1 (graduated) = 18 bytes. We are using slotted pages so each record will have an entry of (record pointer, record length) so each record will need 18 + 8 = 26 bytes. The page footer will also contain the slot count and free space pointer, which takes up 8 bytes. So we can fit floor((1024-8)/(18+8)) = 39 records

iii. (10 points)

For the remainder of the question, assume that the queries are independent of each other; i.e., the 2nd query is run on a copy of the file that has never had the 1st query run on it. And also assume that the buffer is large enough to hold all data and header pages, and starts empty for each question.

A. (3 pt) What is the best case I/O cost of executing the following query? Assume that the minimum record size is 20B and maximum record size is 40B.

```
SELECT sid FROM Student WHERE birthyear = 2001 OR birthyear = 2002;
```

30 I/Os (315 also given full credit)

Best case is that they all fall in the data pages of the first few header pages. 1000 records satisfy the condition. In the best case all of these records are 20B. So each page can fit floor((1024-8)/(20+8)) = 36 records. This means that 1000 records will take up the first ceil(1000/36) = 28 data pages in the best case. Since header pages have 20 entries, we need to read the first 2 header pages. In total, we read 2 header pages + 28 data pages.

B. (3 pt) What is the worst case I/O cost for the following query? Assume that there exists pages that have enough free space. Hint: Insertion of a new record requires checking for uniqueness of its primary key against all existing records in the database.

```
INSERT INTO Student VALUES (186, "Disk", 2005, 0, False);
```

317 I/Os

read all header + data pages then update 1 header and 1 data page => 15 + 300 + 1 + 1 = 317

C. (4 pt) What is the worst case I/O cost for the following query?

```
DELETE FROM Student WHERE birthyear = 1996;
```

630 I/Os

We need to delete 10% of data + update header pages. There are 500 records that satisfy the predicate and there are 300 data pages. Worst case is that we need to delete something from each data page. 15 + 15 * 20 + 300 + 15 = 630 I/Os

5. (14 points) Sorting and Hashing

(a) (7 points) True/False

i. (1 pt) Given N data pages and B buffer pages, it is only possible to finish external merge sort in one pass if B >= N.

- [x] True
- [ ] False

ii. (1 pt) Decreasing the number of buffer pages increases the number of I/Os performed in each pass of the external sorting algorithm.

- [ ] True
- [x] False

False. Regardless of the number of buffer pages every pass performs same 2*N number of IOs.

iii. (1 pt) Eliminating duplicates, grouping records, ordering records, or an upcoming SMJ algorithm are all reasons for sorting.

- [x] True
- [ ] False

True. See lecture 8 slide 3.

iv. (1 pt) You can only build an in-memory hash table if you have B-1 pages as your partition file size, where B is the number of pages in the buffer.

- [ ] True
- [x] False

False, you can build a hash table when the partition is <= B pages.

v. (1 pt) You can always process B*(B-1) pages of data in 2 passes of external hashing algorithm.

- [ ] True
- [x] False

False, we can hash at most B*(B-1) with 2 passes of external hashing (1 partitioning pass + 1 build and probe pass) if the hash function perfectly divides data uniformly. If the data is skewed we may process less pages.

vi. (1 pt) You can hash at most B*(B-1) pages of data with external hashing in 2 partitioning passes.

- [ ] True
- [x] False

False, we can hash at most $B(B-1)^2$ pages of data with 2 partitioning pass of external hashing.

vii. (1 pt) Assuming uniform data spread and no key skew, with B pages of memory and a file size of N pages, hashing a table will always be at least as costly as sorting it.

- [x] True
- [ ] False

(b) (4 points) Sorting Hashing Misc

i. (2 pt) You want to sort N pages with B buffer pages using the external sorting algorithm taught in class. However, assume that right before pass 0, the memory expands by a factor of 2, and right after pass 0 it shrinks by a factor of 4. This means that in pass 0 there are 2*B buffer pages, and for the remaining passes there are B/4 buffer pages. What is the total number of passes (including pass 0) it will take to complete external sorting? Write your answer as an expression in terms of N and B.

1 + ceil(logB/4-1(N/2B))

Pass 0 produces N/2B number of sorted runs. Pass 1 onwards merges B/4-1 sorted runs together at a time.

ii. (2 pt) If you had 10 buffer pages, what is the minimum number of pages you could externally hash to guarantee that you would have to use recursive partitioning? Ignore all assumptions made in earlier questions.

91

Hash B*(B-1) = 90 pages in 1 pass. If we attempt to externally hash 91 pages, even with a uniform hash function, after pass 1 there will be one partition with B+1 pages.

(c) (3 points) Hash table

You would like to hash a table using the external hashing algorithm taught in class. Assume the following information:

- You have B buffer pages, where B is a number greater than or equal to 3.
- The table size is B^3 pages.
- You are using a perfect hash function that uniformly partitions the data.

i. (2 pt) What is the size of each partition after pass 2 of the external hashing algorithm? Write your answer as an expression in terms of B.

$\frac{B^3}{(B-1)^2}$

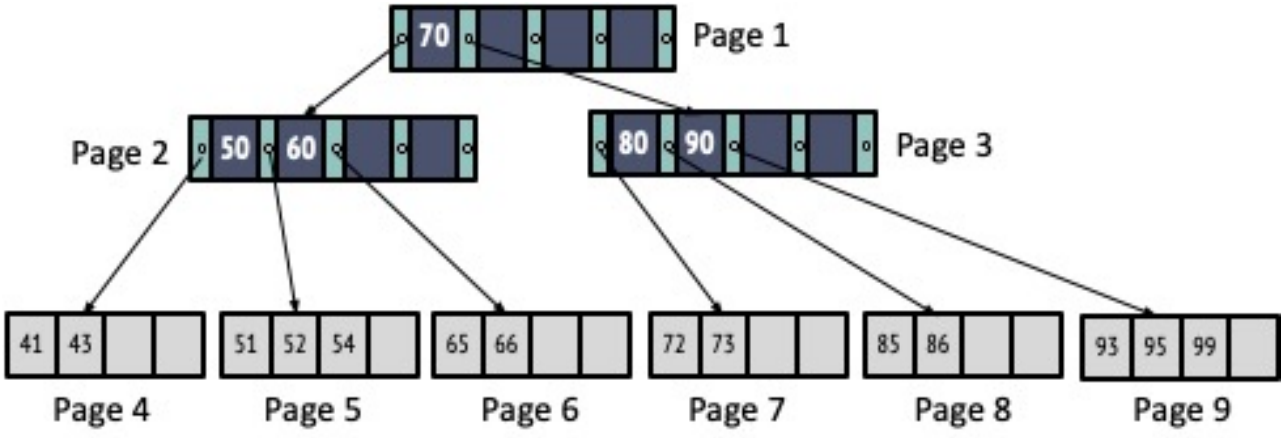
ii. (1 pt) How many passes in total (including the build and probe pass) is needed to hash the table?

4 (5 also given full credit)

Size of each partition after pass 2 is $\frac{B^3}{(B-1)^2}$, which is greater than B. Size of each partition after pass 3 is $\frac{B^3}{(B-1)^3}$ <= B. So we need 3 partitioning passes and 1 last pass to build the hash table.

6. (20 points) Buffer Management

For this problem we will be working with a table of student scores: `Students(id int, score int)` where `id` is the primary key. We use a B+ tree to store data in this table keyed on `id`, with records stored directly in the leaf nodes (i.e., alternative 1), but there are no pointers among the leaf nodes. The tree is organized as follows with 6 data pages total:



For instance, page 4 stores records with ids between 40-49 inclusive, and similarly for pages 5-9.

For simplicity, each of the internal and leaf nodes in the B+ tree takes up exactly 1 page. Assume our buffer pool can fit 4 pages total. We will traverse the tree *depth first*.

To discourage random guessing, for each question below that is worth N points you will get N points for the correct answer, -N/2 for the incorrect answer, and 0 for no answer

For part a assume left to right DFS.

(a) (8 points)

i. (2 pt) We want to use our tree to answer the query `SELECT * FROM Students WHERE id > 55 and id < 75`. We start with an empty buffer. We don't pin any page after it is brought into the buffer pool and hence it can be replaced. Under the *LRU* policy, which pages will be in the buffer pool after we have processed this query?

- ◯ A. 1, 2, 3, 4
- ◯ B. 1, 2, 5, 7
- ◯ C. 2, 3, 5, 6
- ◯ D. 2, 3, 6, 7
- ● E. 3, 5, 6, 7
- ◯ F. 3, 5, 7, 8
- ● G. None of the above.

As clarified during the exam, we will traverse the leftmost branch that is allowed first. We accept two implementations of DFS:

- recursive: visit node N, then visit N's child recursively, then backtrack to N and see if there is another child to visit, etc. We either break off the recursion if we know that we are done, or optionally backtrack all the way back to the root and terminate.
- iterative: visit node N, and while visiting N record all the children of N that should be visited. Then visit each of N's children in a depth first manner. Unlike the recursive approach, when we backtrack to N we do not need to read in the node again as we have previously stored which of N's children to visit.

So for this question we have the following access pattern for the two implementations:

- Recursive: 1, 2, 5, 2, 6, 2, 1, 3, 7, [3, 1], where nodes in [ ] represent our backtracking back to the root
- Iterative: 1, 2, 5, 6, 3, 7

Hence under LRU the following pages will be in the buffer pool after the query is executed:

- Recursive: 1, 2, 3, 7
- Iterative: 3, 5, 6, 7

ii. (2 pt) Suppose we switch to the *MRU* policy, again not pinning any page and starting with an empty buffer. Which pages will be in the buffer pool after we have processed the query above?

- ◯ A. 1, 2, 3, 4
- ● B. 1, 2, 5, 7
- ◯ C. 2, 3, 5, 6
- ◯ D. 2, 3, 6, 7
- ◯ E. 3, 5, 6, 7
- ◯ F. 3, 5, 7, 8
- ● G. None of the above.

Under MRU the following pages will be in the buffer pool after the query is executed:

- Recursive without backtracking to root: 2, 5, 6, 7
- Recursive with backtracking to root: 1, 2, 5, 6
- Iterative: 1, 2, 5, 7

iii. **(2 pt)** We now want to run `SELECT * FROM Students WHERE id > 74` starting with an empty buffer and not pinning any page. Under *LRU*, how many buffer pool *hits* will we get?

- [x] A. 0
- [ ] B. 1
- [x] C. 2
- [x] D. 3
- [ ] E. 4
- [ ] F. None of the above.

We have the following access pattern for the two implementations:

- Recursive: 1, 3, 7, 3, 8, 3, 9, [3, 1], where nodes in [ ] represent our backtrack back to the root
- Iterative: 1, 3, 7, 8, 9

Hence under LRU we get the following:

- Recursive without backtracking to root: 2 hits on page 3
- Recursive with backtracking to root: 3 hits on page 3
- Iterative: 0

iv. **(2 pt)** Suppose we switch to the *MRU* policy, again not pinning any page and starting with an empty buffer. Under *MRU*, how many buffer pool *misses* will we incur? Remember to count each page when it is initially brought into the buffer pool as a miss!

- [ ] A. 0
- [ ] B. 1
- [ ] C. 2
- [ ] D. 3
- [ ] E. 4
- [x] F. None of the above.

Under MRU we get the following:

- Recursive without backtracking to root: 5 misses on pages 1, 3, 7, 8, 9
- Recursive with backtracking to root: 6 misses on pages 1, 3, 7, 8, 3 (again)
- Iterative: 5 misses on pages 1, 3, 7, 8, 9

(b) (6 points)

Raymond comes up with a new idea trying to reduce misses: we will dedicate two of the buffer pool pages for the internal nodes, and the other two for the leaf nodes. Each half will have its own eviction policy.

i. (2 pt) Once again we want run the query `SELECT * FROM Students WHERE id > 55 and id < 75`. We start with an empty buffer. We don't pin any page after it is brought into the buffer pool. If we use LRU for *both* halves of the buffer, which pages will be in the buffer pool after we have processed this query?

- ○ A. 1, 2, 3, 4
- ○ B. 1, 2, 5, 7
- ○ C. 2, 3, 5, 6
- ● D. 2, 3, 6, 7
- ○ E. 3, 5, 6, 7
- ○ F. 3, 5, 7, 8
- ● G. None of the above.

With the same access pattern as before, under LRU we get the following:

- Recursive: 1, 3, 6, 7
- Iterative: 2, 3, 6, 7

ii. (2 pt) Suppose we switch to LRU for the internal nodes, but MRU for the leaf nodes, again not pinning any page and starting with an empty buffer. Which pages will be in the buffer pool after we have processed the query above?

- ○ A. 1, 2, 3, 4
- ○ B. 1, 2, 5, 7
- ○ C. 2, 3, 5, 6
- ○ D. 2, 3, 6, 7
- ○ E. 3, 5, 6, 7
- ○ F. 3, 5, 7, 8
- ● G. None of the above.

With the same access pattern as before, under this new policy we get the following:

- Recursive: 1, 3, 5, 7
- Iterative: 2, 3, 5, 7

iii. (2 pt) Suppose we switch to MRU for the internal nodes, but LRU for the leaf nodes, again not pinning any page and starting with an empty buffer. Which pages will be in the buffer pool after we have processed the query above?

- ○ A. 1, 2, 3, 4
- ○ B. 1, 2, 5, 7
- ○ C. 2, 3, 5, 6
- ● D. 2, 3, 6, 7
- ○ E. 3, 5, 6, 7
- ○ F. 3, 5, 7, 8
- ● G. None of the above.

With the same access pattern as before, under this new policy we get the following:

- Recursive: 2, 3, 6, 7
- Iterative: 1, 3, 6, 7

(c) (6 points)

Joy is not convinced about Raymond's new policy and decides not to use it. Rather than a B+ tree, she instead stores the data as a sorted heap file, with a single header page that links to the first data page, and each data page storing the pointer to the next data page as follows:

Records in each of the data pages are sorted on `id` as in the B+ tree.

She now wants to run the query `SELECT * FROM Students WHERE id > 65 and id < 85` and compare the two designs.

i. (2 pt) Suppose we use *MRU* as the eviction policy, not pinning any page, and starting with an empty buffer. What is the IO cost incurred with running this query using Joy's sorted heap file?

- ◯ A. 1
- ◯ B. 2
- ◯ C. 3
- ◯ D. 4
- ◯ E. 5
- ● F. 6

ii. (4 pt) Suppose the size of the buffer pool is N, the I/O cost of running the query above using the B+ tree is c while the cost using Joy's sorted heap file is 7 (which is not the answer to the question above, by the way). What's the *minimum* value of N such that $c \leq 7$? And under which eviction policy?

- ◯ A. $N = 2$, MRU for B+ tree and sorted file.
- ● B. $N = 2$, LRU for B+ tree and sorted file.
- ◯ C. $N = 3$, MRU for B+ tree and sorted file.
- ◯ D. $N = 3$, LRU for B+ tree and sorted file.
- ◯ E. $N = 4$, MRU for B+ tree and sorted file.
- ◯ F. $N = 4$, LRU for B+ tree and sorted file.
- ◯ G. $N = 5$, MRU for B+ tree and sorted file.
- ◯ H. $N = 5$, LRU for B+ tree and sorted file.
- ◯ I. None of the above.

7. (15 points) SQL and Relational Algebra

The Cal football team recently fired their offensive coach. Pranav wants to be the next coach for the Cal football team and is determined to use data analytics to help the team win more games.

Pranav has access to the following tables below:

```
CREATE TABLE players (
    player_id INT PRIMARY KEY,
    first_name VARCHAR(30),
    last_name VARCHAR(30),
    position VARCHAR(30),
    total_yards FLOAT,
    total_touches INT,
    age INT
);

CREATE TABLE jersey_sales (
    player_id INT PRIMARY KEY,
    last_name VARCHAR(30),
    jersey_number INT,
    sales FLOAT
);

CREATE TABLE big_game_touchdowns (
    player_id INT PRIMARY KEY,
    touchdowns_scored INT,
    yards_gained FLOAT
);
```

The `big_game_touchdowns` table only contains players who scored at least one touchdown in the big game.

(a) (3 points) Truth Values

This data has a few NULL values so Pranav needs to be sure how NULL Boolean logic works in SQL. Help find the following truth values.

i. (1 pt) NULL AND TRUE

- ○ A. True
- ○ B. False
- ● C. Null

ii. (1 pt) NULL AND FALSE

- ○ A. True
- ● B. False
- ○ C. Null

iii. (1 pt) NULL AND NULL

- ○ A. True
- ○ B. False
- ● C. Null

(b) (8 points) SQL Logic

i. (2 pt) Which queries return the same result as `SELECT count(*) FROM players;`?

- ☐ A. `SELECT count(first_name) FROM players;`
- ■ B. `SELECT count(player_id) FROM players;`
- ☐ C. `SELECT count(last_name) FROM players;`
- ☐ D. `SELECT count(age) FROM players;`

ii. (1 pt) Pranav wants to find the best three Wide Receivers to start. Will the following query return the first and last name of the 3 players whose position is "Wide Reciever" and that have the largest number of total yards?

```
SELECT first_name, last_name
FROM players
WHERE position = ''Wide Receiver''
ORDER BY total_yards
LIMIT 3;
```

- ○ True
- ● False

FALSE; no DESC in ORDER BY

iii. (5 pt) Pranav wants to find which players are the most efficient for gaining yards. Select all the SQL queries that return the first and last name of the player(s) that has the highest total yards per total touches ratio.

```
A)
SELECT first_name, last_name
    FROM players
    ORDER BY total_yards/total_touches DESC
    LIMIT 1;

B)
SELECT p1.first_name, p1.last_name
FROM players p1
WHERE p1.total_yards/p1.total_touches >= ALL (
    SELECT  p2.total_yards/p2.total_touches
    FROM players p2
);

C)
SELECT p1.first_name, p1.last_name
FROM players p1
WHERE p1.total_yards/p1.total_touches = ALL (
    SELECT  MAX(p2.total_yards/p2.total_touches)
    FROM players p2
);

D)
SELECT first_name, last_name
    FROM players
    WHERE total_yards/total_touches = MAX(total_yards/total_touches);

E)
SELECT p1.first_name, p1.last_name
FROM players p1
WHERE NOT p1.total_yards/p1.total_touches < ANY (
    SELECT  MAX(p2.total_yards/p2.total_touches)
    FROM players p2
);
```

☐ A

■ B

■ C

☐ D

■ E

☐ F. None of the above

(c) (4 points) Relational Algebra

Fill in the relational algebra expression to find the last names of all players that did not score a touchdown in the big game.

$\pi_{\texttt{(Blank 1)}}$(players `___(Blank2)___`($\pi_{\texttt{(Blank 3)}}$`table1` - $\pi_{\texttt{(Blank 4)}}$`table_2`)

Assume `table1` is the table name corresponding to `Blank 3` and `table2` is the table name corresponding to `Blank 4`.

i. (1 pt) `Blank 1`

- (●) A. players.last_name
- ( ) B. players.id
- ( ) C. players.first_name

ii. (1 pt) `Blank 2`

- (●) A. Join Symbol
- ( ) B. Union Symbol
- ( ) C. Intersect Symbol

iii. (1 pt) Blank 3

- (●) A. players.player_id
- ( ) B. big_game_touchdowns.player_id
- ( ) C. big_game_touchdowns.last_name
- ( ) D. players.last_name

iv. (1 pt) Blank 4

- ( ) A. players.player_id
- (●) B. big_game_touchdowns.player_id
- ( ) C. big_game_touchdowns.last_name
- ( ) D. players.last_name

8. (29 points) USA Wins 0-0! SIUU! (it's called soccer)

(a) (8 points)

Vik, an avid USA and Portugal fan, has spent a significant chunk of his recent days watching the World Cup. He decides to organize team and player statistics to help him bet on the games!

Assume for the following questions we are given the following schemas:

```
CREATE TABLE Teams (
    team_id INTEGER PRIMARY KEY,
    country CHAR(30) NOT NULL,
    ranking INTEGER NOT NULL,
    group_stage_points INTEGER NOT NULL,
    goals_scored INTEGER NOT NULL,
    goals_conceded INTEGER NOT NULL,
    in_knockouts BOOLEAN
);

CREATE TABLE Players(
    player_id INTEGER PRIMARY KEY,
    team INTEGER NOT NULL,
    name CHAR(30) NOT NULL,
    goals INTEGER NOT NULL,
    assists INTEGER,
    FOREIGN KEY (team) REFERENCES Teams (team_id)
);
```

- Integers are 4 bytes and pointers are 4 bytes.
- **Record headers contain a bitmap**. Bitmaps should be as small as possible, rounded up to the nearest byte.
- Each data page is 1 KB (1 KB = 1024 B).
- Assume we use the unpacked format for pages and that the page header contains the record count

i. (1 pt) What is the maximum size of a record from the Team table?

52. 1 byte for the bitmap, 4 bytes for team_id, 30 bytes for country, 4 bytes for ranking, 4 bytes for group_stage_points, 4 bytes for goals_scored, 4 bytes for goals_conceded, 1 byte for in_knockouts

ii. (1 pt) What is the minimum size of a record from the Team table?

52. Even though we can nullify in_knockouts, we must still reserve space for the field in the record.

iii. (1 pt) What is the maximum size of a record from the Players Table?

47

iv. **(1 pt)** What is the minimum size of a record from the Players Table?

47

v. **(2 pt)** What is the maximum number of Team records that can fit onto one data page?

19

Each record is 52 bytes = 416 bits. The unpacked page format includes an additional bit per record in the header. Therefore, the overhead per record is 417 bits. Each page is 1024 bytes, 4 fo which are reserved for the record count in the header. Therefore the total usable page data is 1020 bytes = 8160 bits.

Floor(8160 / 417) = 19

vi. **(2 pt)** What is the maximum number of Players records that can fit onto one data page?

21

Each record is 47 bytes = 376 bits. The unpacked page format includes an additional bit per record in the header. Therefore, the overhead per record is 377 bits. Each page is 1024 bytes, 4 of which are reserved for the record count in the header. Therefore the total usable page data is 1020 bytes = 8160 bits.

Floor(8160 / 377) = 21

(b) (6 points)

Vik's friend Lucy wants to help him in his betting aspirations and decides to run a query to aggregate important information about the teams and players.

Here are the same two tables from the previous part, copied down for your convenience:

```
CREATE TABLE Teams (
    team_id INTEGER PRIMARY KEY,
    country CHAR(30) NOT NULL,
    ranking INTEGER NOT NULL,
    group_stage_points INTEGER NOT NULL,
    goals_scored INTEGER NOT NULL,
    goals_conceded INTEGER NOT NULL,
    in_knockouts BOOLEAN
);
```

```
CREATE TABLE Players(
    player_id INTEGER PRIMARY KEY,
    team INTEGER NOT NULL,
    name CHAR(30) NOT NULL,
    goals INTEGER NOT NULL,
    assists INTEGER,
    FOREIGN KEY (team) REFERENCES Teams (team_id)
);
```

- Integers are 4 bytes and pointers are 4 bytes.
- Each data page is 1 KB (1 KB = 1024 B).
- Assume we use the unpacked format for pages and that the page header contains the record count

Now assume we are given the following query to execute:

```
SELECT a.country, a.team_id, b.team_id, a.group_stage_points,
    a.goals_scored - a.goals_conceded AS goal_diff,
    p.player_id, p.goals, p.name, a.in_knockouts
FROM Teams as a, Teams as b, Players as p
WHERE a.team_id = b.team_id AND p.team = a.team_id AND a.in_knockouts = True
ORDER BY a.group_stage_points
```

- The selectivity of Teams.in_knockouts = True is **0.5**
- Ignore your answers to the previous questions and assume there are 4 records per page of Teams and 8 total pages of Teams
- **Assume records DO NOT have bitmaps** (this is the same as if all fields are NOT NULL)
- Assume in the Players table there are exactly 26 unique players with the same team_id, per team (i.e., for every team in the Teams table there are 26 matching players in the Players table)

i. (2 pt) How many total records does the Players table contain?

832

32 total records in Teams i.e., 32 unique teams (with their own team_ids). Since we are given 26 players per team in the Players table, that is 32 * 26 or 832 total players or 832 total records in the Players table

ii. (4 pt) How many total pages does the output of the above query produce?

A. Joining Teams with itself does not increase the total number of records, since we are joining on the primary key. The same applies to joining Teams with Players - each player just matches with the corresponding row of the team the player is on and that row is just appended to the players row, forming a larger row but not creating any new rows. Thus the total number of records after all the joins is the same as the total number of records in Players which is 832. Since our selectivity condition is $\frac{1}{2}$, The total number of output records of our query is $832/2 = 416$

Now, we have to calculate the size of each record of the output table. This is 30 * 2 + 4 * 6 + 1 = 85 bytes (no bitmap)

Using the same logic as the above questions, the total overhead per record is 85 * 8 + 1 bits = 681 bits. Also, we have 1020 usable bytes per page which is 8160 bits.

Overall, we can fit Floor(8160/681) records per page which is 11 full records per page. The total number of pages required is Ceil(416/11) which is 38.

(c) (4 points)

Here are the prior tables and queries copied down for your convenience:

```
CREATE TABLE Teams (
    team_id INTEGER PRIMARY KEY,
    country CHAR(30) NOT NULL,
    ranking INTEGER NOT NULL,
    group_stage_points INTEGER NOT NULL,
    goals_scored INTEGER NOT NULL,
    goals_conceded INTEGER NOT NULL,
    in_knockouts BOOLEAN
);
```

```
CREATE TABLE Players(
    player_id INTEGER PRIMARY KEY,
    team INTEGER NOT NULL,
    name CHAR(30) NOT NULL,
    goals INTEGER NOT NULL,
    assists INTEGER,
    FOREIGN KEY (team) REFERENCES Teams (team_id)
);
```

```
SELECT a.country, a.team_id, b.team_id, a.group_stage_points,
    a.goals_scored - a.goals_conceded AS goal_diff,
    p.player_id, p.goals, p.name, a.in_knockouts
FROM Teams as a, Teams as b, Players as p
WHERE a.team_id = b.team_id AND p.team = a.team_id AND a.in_knockouts = True
ORDER BY a.group_stage_points
```

For this question, assume the following information (ignore all other information from previous questions and answers):

- Teams has 50 records per page and 50 total pages
- Players has 100 records per page and 100 total pages
- The selectivity of Teams.in_knockouts = True is **0.5**
- Assume in the Players table that there are exactly 26 unique players with the same team_id, per team (ie for every team in the Teams table there are 26 matching players in the Players table)
- Assume an alternative 2 unclustered B+ tree index of height 3 on Player.player_id
- Assume an alternative 1 B+ tree index of height 2 on Teams.in_knockouts
- Assume all entries with the same key fit on the same leaf node (this is not meant to confuse you, this is meant to simplify assumptions and calculations)

i. (2 pt) We wish to join Teams with Players using Index Nested Loop Join as given in the above query.

What is the best I/O cost of Teams INLJ Players (in that order)

Use the alternative 2 index on Player.player_id. Also apply selection on Teams after the initial read.

[T] + Rho(T) * Ceil([T]/2) * (cost to find matching Players tuples) = 50 + 50 * 25 * (4 + 26) = 37550

ii. (2 pt) What is the best I/O cost of Players INLJ Teams (in that order)

Without a corresponding index, INLJ reduces to a SNLJ. Additionally, we are forced to apply selectivity on the fly every iteration, so we are unable to derive I/O benefits.

[P] + Rho(P) * [P] * [T] = 100 + 100 * 100 * 50 = 500100

(d) (2 points)

Here are the prior tables and queries copied down for your convenience:

```
CREATE TABLE Teams (
    team_id INTEGER PRIMARY KEY,
    country CHAR(30) NOT NULL,
    ranking INTEGER NOT NULL,
    group_stage_points INTEGER NOT NULL,
    goals_scored INTEGER NOT NULL,
    goals_conceded INTEGER NOT NULL,
    in_knockouts BOOLEAN
);
```

```
CREATE TABLE Players(
    player_id INTEGER PRIMARY KEY,
    team INTEGER NOT NULL,
    name CHAR(30) NOT NULL,
    goals INTEGER NOT NULL,
    assists INTEGER,
    FOREIGN KEY (team) REFERENCES Teams (team_id)
);
```

```
SELECT a.country, a.team_id, b.team_id, a.group_stage_points,
    a.goals_scored - a.goals_conceded AS goal_diff,
    p.player_id, p.goals, p.name, a.in_knockouts
FROM Teams as a, Teams as b, Players as p
WHERE a.team_id = b.team_id AND p.team = a.team_id AND a.in_knockouts = True
ORDER BY a.group_stage_points
```

For this question, assume the following information (ignore all other information from previous questions and answers):

- [Teams] = 40 with 5 records per page
- [Players] = 720 with 5 records per page
- B = 8
- The selectivity of Teams.in_knockouts = True is **0.5**

Assume we now want to join Teams and Players using Sort Merge Join as given in the above query.

i. (2 pt) What is the best I/O cost of a Sort Merge Join between Teams and Players

A.

Cost of sorting Players: 720 + 360 + (2 * 2) * 360 = 2520 (read in the entire Players table, but push down the selection on Teams.in_knockouts before pass 0 of sorting)

Cost of sorting Teams: 2 * 2 * 40 = 160

Runs of Players: 7; Runs of Teams: 5; Since Runs of Teams + 1 <= 8 - 1, we can utilize the optimization to save 2 * |Teams| I/Os

Total cost: 2520 + 160 + 360 + 40 - (2 * 40) = 3000.

(e) (9 points)

Here are the prior tables and queries copied down for your convenience:

```
CREATE TABLE Teams (
    team_id INTEGER PRIMARY KEY,
    country CHAR(30) NOT NULL,
    ranking INTEGER NOT NULL,
    group_stage_points INTEGER NOT NULL,
    goals_scored INTEGER NOT NULL,
    goals_conceded INTEGER NOT NULL,
    in_knockouts BOOLEAN
);
```

```
CREATE TABLE Players(
    player_id INTEGER PRIMARY KEY,
    team INTEGER NOT NULL,
    name CHAR(30) NOT NULL,
    goals INTEGER NOT NULL,
    assists INTEGER,
    FOREIGN KEY (team) REFERENCES Teams (team_id)
);
```

```
SELECT a.country, a.team_id, b.team_id, a.group_stage_points,
    a.goals_scored - a.goals_conceded AS goal_diff,
    p.player_id, p.goals, p.name, a.in_knockouts
FROM Teams as a, Teams as b, Players as p
WHERE a.team_id = b.team_id AND p.team = a.team_id AND a.in_knockouts = True
ORDER BY a.group_stage_points
```

i. (3 pt) For this question, assume the following information (ignore all other information from previous questions and answers):

- Teams has 50 pages with 100 records each
- Players has 40 pages with 100 records each
- The selectivity of Teams.in_knockouts = True is **0.5**
- Unclustered, height 2, Alt 3 B+ tree built on Players.team with 20 leaf nodes

Estimate the I/O Cost of an index scan on Players.team as our access plan for Players in Pass 1

2 I/Os for root to level above leaves 20 I/Os for all leaves 40 * 100 I/Os for all records

Total = 4022 I/Os

ii. (2 pt) For this question, assume the following information (ignore all other information from previous questions and answers):

```
A: I/Os: 50; Full scan on Players
B: I/Os: 50; Index Scan on Players.player_id
C: I/Os: 70; Index Scan on Players.team
D: I/Os: 90; Index Scan on Teams.group_stage_points
E: I/Os: 40; Index Scan on Teams.country
F: I/Os: 40; Index scan on Teams.team_id
G: I/Os: 40; Full Scan on Teams
H: I/Os: 70; Index Scan on Teams.in_knockouts
I: I/Os: 30; Index Scan on Teams.team_id
```

With the cost-based optimizer that we learned, which of the following plans will move forward from Pass 1?

- [x] A
- [x] B
- [x] C
- [x] D
- [ ] E
- [ ] F
- [ ] G
- [ ] H
- [x] I

A: min cost*

B: min cost*

C: interesting order

D: interesting order

I: min cost

*Selecting A and not B, B and not A, or both A and B all received full credit as both A and B provide a min-cost scan of Players

iii. (4 pt) For this question, assume the following information (ignore all other information from previous questions and answers):

```
A: I/Os: 500; Players BNLJ Players
B: I/Os: 600; Players SMJ Players (on team)
C: I/Os: 350; Players GHJ Players
D: I/Os: 750; Teams BNLJ Players
E: I/Os: 700; Players BNLJ Teams
F: I/Os: 850; Teams SMJ Players (on team_id)
G: I/Os: 600; Players GHJ Teams
H: I/Os: 750; Teams BNLJ Teams
I: I/Os: 800; Teams SMJ Teams (on country)
J: I/Os: 750; Teams GHJ Teams
K: I/Os: 900; Teams SMJ Teams (on group_stage_points)
L: I/Os: 750; Teams SMJ Teams (on team_id)
M: I/Os: 700; Teams SMJ Teams (on in_knockouts)
```

With the cost-based optimizer that we learned, which of the following plans will move forward from Pass 2?

☐ A

☐ B

☐ C

☐ D

☐ E

■ F

■ G

☐ H

☐ I

☐ J

■ K

■ L

■ M

F: interesting order

G: min cost

K: interesting order

L: interesting order

M: min cost

3000. (10 points) PQP

(a) (7 points) Parallel Joins

i. (4 points)

We want to perform parallel grace hash join on relations R and S. Both relations are 1000 pages and each page is 2 KB. Initially R is round-robin partitioned across 10 machines, and S is hash partitioned across the same 10 machines, but we do not know the hash function used for partitioning S. Assume all hash functions are uniform and all machines have 11 buffer pages.

A. (1 pt) Can we use different hash functions for partitioning R and S?

○ Yes

● No

B. (1 pt) What is the expected network cost of partitioning R in KB?

1800

C. (1 pt) What is the expected network cost of partitioning S in KB?

1800

D. (1 pt) How many passes through relation R are needed to perform the join?

4

There is 1 pass for hash partitioning R, S across the 10 machines. Now each machine has 100 pages of R, S. After a partitioning pass, each machine has partitions with 10 pages of R,S. After another partitioning pass, each machine has partition with 1 page of R,S. It is at this point we can build and probe, causing another pass. This gives 4 passes in total.

ii. (3 points) Parallel SMJ

Now suppose we want to perform a parallel sort merge join with the first relation being the result of joining relations R and S as described in (a), and the second relation being relation T. Relation T is 100 pages, with each page being 2 KB. Suppose relation T is already uniformly range-partitioned across 5 machines separate from the ones above.

A. (1 pt) Can tuples be pipelined concurrently from one operator to another throughout the entire query plan (pipeline parallelism)?

- ○ A. No, since grace hash join is a pipeline breaker.
- ● B. No, since sort merge join is a pipeline breaker.
- ○ C. No, since the network communication cost is infeasible.
- ○ D. Yes.

B. (2 pt) How many passes of relation T are needed to perform the join (including the range partitioning)? Assume that the column we are joining on has no duplicates and we perform no optimization.

4

(b) (3 points) Trivia

i. (2 pt) Suppose we want to multiply matrices A and B. We partition matrix B evenly by its columns into B1 and B2. We compute A*B1 and A*B2 on separate machines using a single join operator and concatenate the results at the end to get A*B. What type of parallelism schemes are used here? Select all that apply.

- [x] A. Intra-operator parallelism.
- [ ] B. Pipeline parallelism.
- [x] C. Sharding.
- [x] D. Replication.

ii. (1 pt) Range partitioning is usually easier for smaller datasets than larger datasets.

- (●) Yes.
- ( ) No.

3001. (28 points) Deadlock Avoidable RookieDB

Justin is thinking about designing a deadlock-free concurrency manager, and he decided to do so through deadlock avoidance. He wished to implement both the **Wait-Die** and **Wound-Wait** protocols. Help him implement the code!

(a) (13 points) Build stage

We first modify our `TransactionContext` so as to reflect the priority of a transaction. Assume we have access to these methods in `TransactionContext`

```
/**
 * Return the priority value of the current transaction.
 * The priority is defined by the age of the transaction.
 * The larger the returned value is, the higher priority it has.
 */
public int getPriority() {
    Return time.now() - this.startTime();
}

/* Abort the current transaction */
protected void startRollback() {
    /* implementation hidden */
}
```

and some useful classes and methods are reproduced/modified as below

```
public class Lock {
    public ResourceName name;
    public LockType lockType;
    public TransactionContext transaction;
    ... // additional fields
}

public enum LockType {
    /* This method checks whether lock types A and B are compatible
     * with each other. */
    public static boolean compatible(LockType a, LockType b) {
        /* implementation hidden */
    }
}

private class ResourceEntry {
    /* Adds `request` to the front of the queue if addFront is true,
     * or to the end otherwise. */
    public void addToQueue(LockRequest request, boolean addFront) {
        /* implementation hidden */
    }

    /* Check if `lockType` is compatible with preexisting locks. Allows conflicts
     * for locks held by transaction with id `except`, which is useful when
     * a transaction tries to replace a lock it already has on the resource. */
    public boolean checkCompatible(LockType lockType, long except) {
        /* implementation hidden */
}
```

i. (6 points) Wait-die acquire

Now we will implement the wait-die protocol's `acquire` method. Note that in the case where the

current transaction t_i wants a lock on a resource where multiple transactions t_j's hold locks on as well, t_i has to check with every other t_j to see if it should wait.

```
public void waitdieAcquire(TransactionContext transaction, ResourceName name,
            LockType lockType) throws DuplicateLockRequestException {
    ... // error checking

    ResourceEntry entry = getResourceEntry(name);

    Lock newLock = new Lock(name, lockType, transaction);
    Request request = new LockRequest(transaction, newLock);
    if (entry.checkCompatible(lockType, transaction.getTransNum()) &&
        entry.waitingQueue.isEmpty()) {
        entry.grantOrUpdateLock(newLock);
    } else {
        for (Lock lock: entry.locks) {
            if (__(A)__) {
                __(B)__;
                return;
            }
        }
        __(C)__;
    }
}
```

A. (2 pt) Fill in the code that should replace (A)

```
transaction.getPriority() < lock.transaction.getPriority()
```

B. (2 pt) Fill in the code that should replace (B)

```
transaction.startRollback()
```

C. (2 pt) Fill in the code that should replace (C)

```
entry.addToQueue(request, false)
```

ii. (7 points) Wound-wait acquire

- We only abort a transaction when necessary, i.e., if a transaction t_j is compatible with the current transaction, we don't abort it.
- Out of all the incompatible transactions t_j's, if any of them cannot be aborted, the current transaction has to wait

```
public void woundwaitAcquire(TransactionContext transaction, ResourceName name,
            LockType lockType) throws DuplicateLockRequestException {
    ... // error checking

    ResourceEntry entry = getResourceEntry(name);

    Lock newLock = new Lock(name, lockType, transaction);
    Request request = new LockRequest(transaction, newLock);
    if (entry.checkCompatible(lockType, transaction.getTransNum()) &&
        entry.waitingQueue.isEmpty()) {
        entry.grantOrUpdateLock(newLock);
    } else {
        boolean waits = false;
        for (Lock lock: entry.locks) {
            if (__(A)__) {
                if (transaction.getPriority() > lock.transaction.getPriority()) {
                    __(B)__;
                } else {
                    __(C)__;
                }
            }
        }
        if (waits) {
            __(D)__;
        }
    }
}
```

A. (2 pt) Fill in the code that should replace (A)

```
!LockType.compatible(lockType, lock.lockType)
```

`entry.checkCompatible` is different from `LockType.compatible`. When the current lock is incompatible with the existing locks, we need to check with every other lock to see if each pair is compatible.

B. (2 pt) Fill in the code that should replace (B)

```
lock.transaction.startRollback()
```

C. (1 pt) Fill in the code that should replace (C)

```
waits = true
```

D. (2 pt) Fill in the code that should replace (D)

```
entry.addToQueue(request, false)
```

(b) (15 points) Use stage

Now our database is guaranteed to not get into a deadlock, let's run some transactions on both the wait-die database and wound-wait database!

- The first two schedules will be using the **wait-die** protocol we implemented
- The third schedule will be using the **wound-wait** protocol we implemented

For all questions in this section, assume that:

- if a transaction is aborted, it will roll back and release all its locks before the next timestamp
- if a transaction is blocked, the execution will switch to the next available transaction in terms of the timestamp
- Once a lock is released, it will process its queue, similar to the project
- Priority is determined by age

i. (3 points) A very bad schedule

Justin issued a very bad schedule of transactions, which should create a deadlock on a normal concurrency manager. We have only one resource: `table`. The schedule is listed as follows:

This schedule uses **wait-die** protocol

	Timestamp	T1	T2
1		BEGIN TXN	
2			BEGIN TXN
3		S(table)	
4			S(table)
5		X(table)	
6			X(table)

A. (1 pt) What transaction(s) is aborted after the above schedule is executed?

- ☐ A. T1
- ■ B. T2
- ☐ C. None

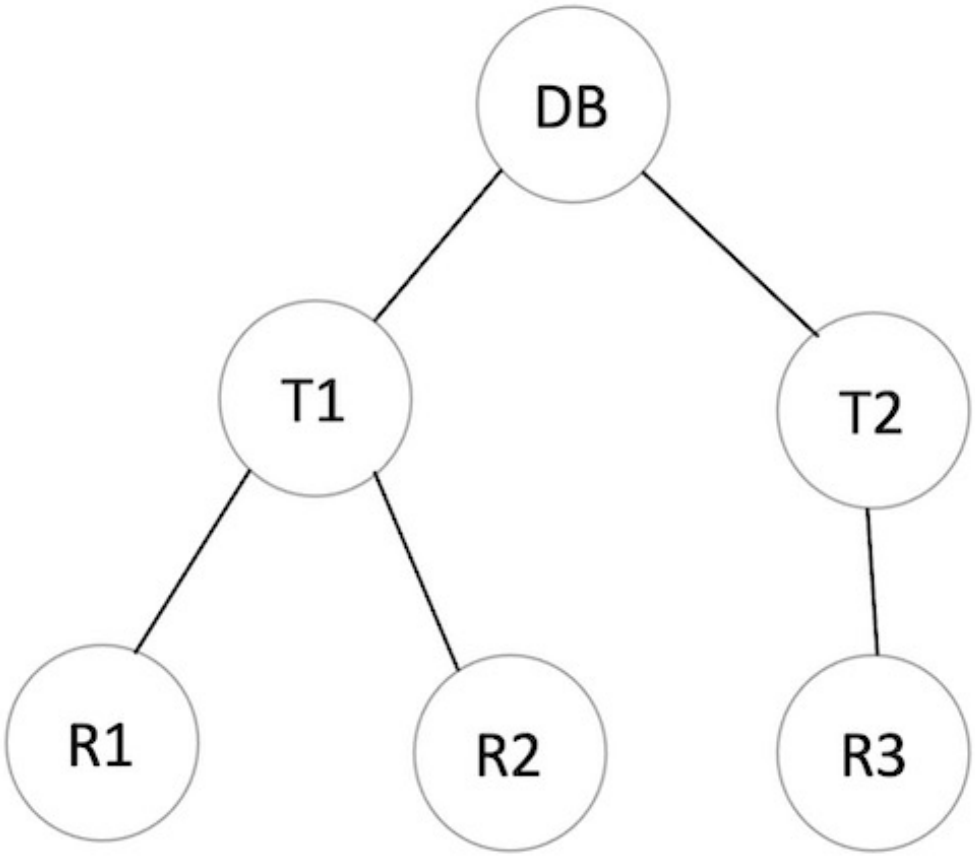
B. (1 pt) What lock does T1 hold on `table` after all timestamps? If none, write "None".

- ○ A. S
- ● B. X
- ○ C. NL

C. (1 pt) What lock does T2 hold on `table` after all timestamps?

- ○ A. S
- ○ B. X
- ● C. NL

ii. (5 points) An interesting schedule



This schedule uses **wait-die** protocol.

	Timestamp	T1	T2	T3
1	BEGIN TXN			
2		BEGIN TXN		
3				BEGIN TXN
4	IS(DB)			
5	IS(T1)			
6	S(R2)			
7				IS(DB)
8				S(T2)
9			IX(DB)	
10			IX(T2)	
11			X(R3)	

A. (2 pt) What transaction(s) is aborted after the above schedule is executed?

- ☐ A. T1
- ☐ B. T2
- ☐ C. T3
- ■ D. None

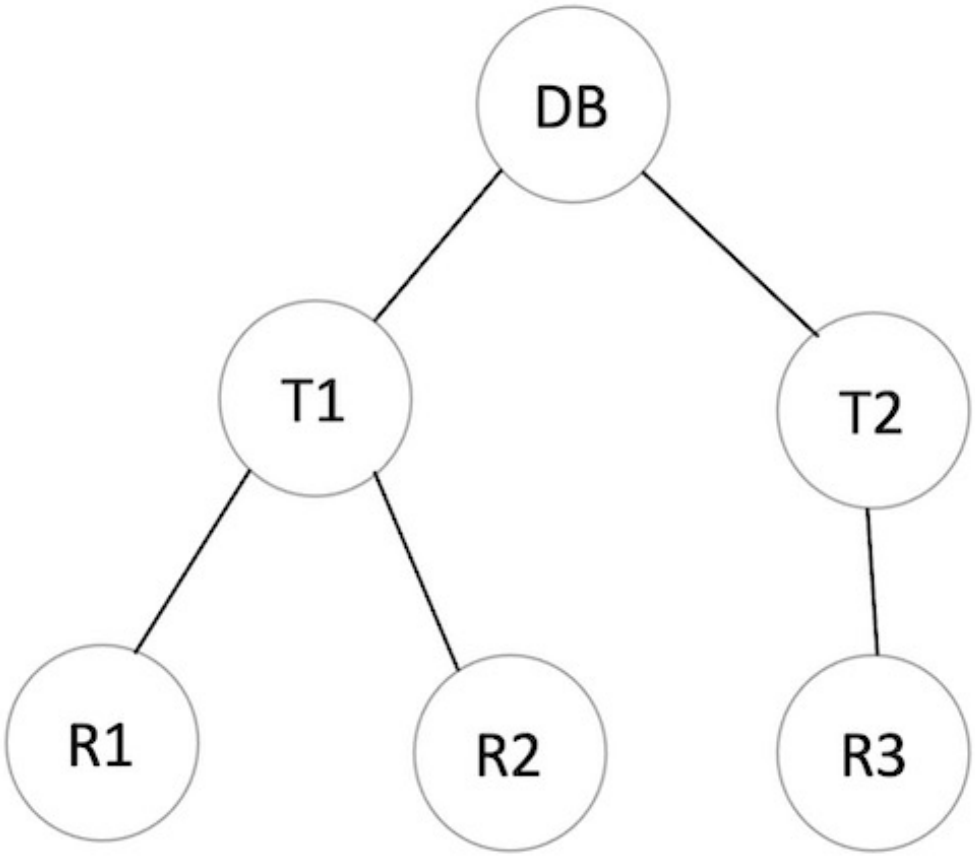
B. (2 pt) One transaction gets blocked due to having to wait for another transaction. What transaction is it?

- ○ A. T1
- ● B. T2
- ○ C. T3
- ○ D. T186

C. (1 pt) At what timestamp did this transaction get blocked?

10

iii. (3 points) Wound-wait schedule I



This schedule uses **wound-wait** protocol.

	Timestamp	T1	T2	T3
1	BEGIN TXN			
2		BEGIN TXN		
3				BEGIN TXN
4	IX(DB)			
5	IX(T1)			
6	X(R1)			
7		IX(DB)		
8		IX(T2)		
9				IS(DB)
10				IS(T1)
11				S(R2)
12	S(DB)			

A. (2 pt) What transaction(s) is aborted after the above schedule is executed?

- ☐ A. T1
- ■ B. T2
- ☐ C. T3
- ☐ D. None

B. (1 pt) What lock does T1 hold on DB after going through the schedule? If none write "None".

SIX

iv. (4 pt) Sounds like our new RookieDB is very powerful in that we can write arbitrary schedules without worrying about the issue of deadlock.

Give one disadvantage of this new concurrency manager.

We are aborting too many transactions. Or: the concurrency will be reduced.

No more questions.