CS 245 Final Exam
Spring 2019 (Solutions)

- Please read all instructions (including these) carefully.
- There are seven problems, some with multiple parts, for a total of 100 points. You have **2.5 hours (150 minutes)** to work on the exam.
- The exam is open notes. You may use a laptop with all communication facilities (Wi-Fi, Bluetooth, etc.) disabled.
- Please write your answers in the space provided on the exam, and clearly mark your solutions. You may use the backs of the exam pages as scratch paper. Please do not use any additional scratch paper.
- Solutions will be graded on correctness and clarity. For the long-answer problems, please show your intermediate work. Each problem has a relatively simple to explain solution, and we may deduct points if your solution is much more complex than necessary. Partial solutions will be graded for partial credit.

NAME: ____

SUID: ____

In accordance with both the letter and spirit of the Honor Code, I have neither given nor received assistance on this examination.

SIGNATURE: ____

Grading: P1=20pt, P2=17pt, P3=12pt, P4=14pt, P5=10pt, P6=15pt, P7=12pt, total=100pt.

Problem 1: Short Answer Questions (20 points)

1. (2 points) Which of these statements about serializable schedules is true? *(Circle one answer.)*

(a) Every serializable schedule is recoverable.

(b) Every serializable schedule contains no conflicting actions.

[✓ (c) Every 2PL schedule is serializable.]

(d) None of the above.

2. (2 points) Which of these statements about recoverable schedules is true? *(Circle one answer.)*

(a) Every recoverable schedule is serializable.

[✓ (b) In a recoverable schedule, if a transaction T commits, then any other transaction that T read from must also have committed.]

(c) In a recoverable schedule, no transaction will ever be aborted because a transaction that it read from has aborted.

(d) None of the above.

3. (2 points) In which of the following situations is optimistic concurrency control with validation likely to perform better than locking with 2PL? *(Circle one answer.)*

(a) A high-contention workload where all the transactions need to update a single record.

[✓ (b) A read-mostly workload, where most transactions just read a small number of data items, and a few transactions write data items.]

(c) A distributed database where all the transactions need to read and write objects on multiple servers.

(d) All of the above.

4. (2 points) What is the main benefit of extendible hashing over a traditional hash table based on a single array of buckets? *(Circle one answer.)*

(a) Extendible hashing requires fewer random disk I/Os to access each item.

(b) Extendible hashing requires less storage space for the hash table data structure.

[✓ (c) Extendible hashing reduces the I/O needed to update the data structure when we fill a hash bucket.]

(d) Extendible hashing produces fewer hash collisions.

5. (2 points) Which of these statements about deadlock are true? *(Fill in all that are true.)*

☐ If all transactions use two-phase locking, they cannot deadlock.

[✓ ☐ Once two transactions deadlock, one of them must be aborted to maintain correctness.]

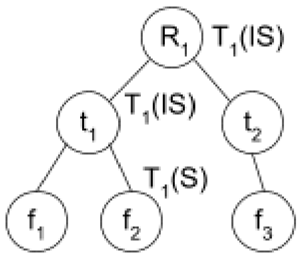
☐ Systems that support update locks (S, X and U modes) cannot deadlock.

[✓ ☐ Validation based concurrency control schemes cannot deadlock.]

6. (2 points) Why is the increment lock mode (I) incompatible with both the S and X modes?

Transactions holding an item in increment mode can add values to it, so it is unsafe for other transactions to either read or write the object because this could change the results they observe.

7. (2 points) In the database below with hierarchical locking, transaction T_1 holds some locks. Which of the following transactions can still acquire the locks they need to run? *(Fill in all that can; assume that only one of these candidate transactions would run at a time.)*



☐ T_2: write t_1.

[✓ ☐ T_3: write t_2.]

[✓ ☐ T_4: read f_2 and write f_1.]

[✓ ☐ T_5: insert a child node under t_2.]

8. (2 points) Which of the following statements about data lakes vs. traditional analytical data warehouses are true? *(Fill in all that are true.)*

☐ Data lakes provide stronger transaction isolation levels than data warehouses.

☐ Data lakes separate storage resources from compute resources, so that they can be scaled up and down independently.

☐ Data lakes store data in standard formats that are accessible by many independently developed compute engines, such as TensorFlow, Apache Spark and Hadoop.

☐ Data lakes enable indexing schemes that data warehouses can't support, such as Z-order.

9. (2 points) Company X stores private health data about its customers, but it only allows its data analysts to query this data through a differentially private query system like PINQ. What security threats does differential privacy protect Company X against? *(Fill in all that are true.)*

☐ Differential privacy ensures that administrators with full access to Company X's servers cannot access any customer's private data.

☐ Differential privacy ensures that a data analyst with a limited privacy budget for her queries can only learn a bounded amount of information about each customer's data.

☐ Differential privacy ensures that if two analysts with separate privacy budgets combine their results, they can't learn more information than one analysts would have on her own.

☐ Differential privacy ensures that aggregate queries such as counts do not reveal any information about the database.

10. (2 points) Recall that *consensus* protocols require that the sets of nodes that participate in different operations always have a non-empty intersection. Why can primary-backup systems be viewed as a form of consensus, given this definition?

In primary-backup systems, all operations must go through the primary node, so this one node is a participant for every operation.

Problem 2: Transaction Schedules (17 points)

a) (4 points) For each of the following schedules of reads and writes done by transactions, indicate whether the schedule is conflict serializable, serializable, and/or serial by filling in the appropriate boxes. We use the notation from lecture, where $r_i(A)$ and $w_i(A)$ mean that transaction T_i reads and writes object A respectively.

i) $r_1(A)\ r_2(A)\ w_1(A)\ r_2(B)$

[✓ Conflict serializable] [✓ Serializable] ☐ Serial

ii) $r_1(A)\ w_1(A)\ r_2(A)\ w_2(B)$

[✓ Conflict serializable] [✓ Serializable] [✓ Serial]

iii) $w_1(A)\ w_2(A)\ w_1(A)\ w_2(A)\ w_1(A)$

☐ Conflict serializable [✓ Serializable] ☐ Serial

iv) $w_1(A)\ w_2(A)\ w_1(B)\ w_2(B)\ w_1(B)$

☐ Conflict serializable ☐ Serializable ☐ Serial

b) (1 point) Draw the precedence graph for the schedule $w_1(A)\ w_2(A)\ r_3(B)\ w_3(B)\ r_1(B)$:

$$\begin{tikzcd}
T3 \arrow[r] & T1 \arrow[r] & T2
\end{tikzcd}$$

c) (6 points) For each of the following schedules of lock, read and write actions done by transactions, fill the appropriate boxes to indicate whether each of the transactions is well-formed and 2PL, and whether the schedule is legal. We use the notation from lecture, where $r_i(A)$, $w_i(A)$, $l_i(A)$, and $u_i(A)$ mean that transaction T_i reads, writes, locks or unlocks object A respectively.

i) $l_1(A)\ l_2(B)\ w_1(A)\ u_1(A)\ r_2(B)\ w_2(B)\ u_2(B)$

[✓ T_1 well-formed] [✓ T_2 well-formed] [✓ T_1 2PL] [✓ T_2 2PL] [✓ Legal]

ii) $l_1(A)\ w_1(A)\ l_2(B)\ w_2(B)\ l_1(B)\ w_1(B)\ u_1(B)\ u_1(A)$

[✓ T_1 well-formed] ☐ T_2 well-formed [✓ T_1 2PL] [✓ T_2 2PL] ☐ Legal

(Since T_2 is not well-formed, we also accepted answers saying it's not 2PL.)

iii) $l_1(A)\ w_1(A)\ l_2(B)\ w_2(B)\ u_2(B)\ u_1(A)\ l_2(A)\ w_2(A)\ u_2(A)$

[✓ T_1 well-formed] [✓ T_2 well-formed] [✓ T_1 2PL] ☐ T_2 2PL [✓ Legal]

d) (3 points) Suppose that each of the sequences of actions below is followed by an abort action for transaction T_1. Which of the other transactions would need to be rolled back? (Fill in the appropriate boxes to indicate your answer.)

i) $w_1(A)\ r_2(A)\ w_1(B)\ r_3(B)\ r_4(C)$

[✓ T_2 rolls back] [✓ T_3 rolls back] ☐ T_4 rolls back

ii) $w_1(A)\ w_2(A)\ w_1(B)\ r_3(B)\ r_4(A)\ r_4(C)$

☐ T_2 rolls back [✓ T_3 rolls back] ☐ T_4 rolls back

iii) $r_1(A)\ w_1(B)\ r_2(A)\ r_2(B)\ r_3(A)\ r_4(B)$

[✓ T_2 rolls back] ☐ T_3 rolls back [✓ T_4 rolls back]

e) (3 points) For each of the following schedules of read, write, commit and abort actions done by transactions, indicate whether they are recoverable, avoids-conflicting-rollback (ACR) and/or strict by filling in the appropriate boxes. We use the notation from lecture, where $r_i(A)$, $w_i(A)$, c_i and a_i mean that transaction T_i reads A, writes A, commits or aborts respectively.

i) $w_1(A)\ r_2(A)\ w_1(B)\ r_2(B)\ c_1\ c_2$

[✓ Recoverable] ☐ ACR ☐ Strict

ii) $w_1(A)\ w_1(B)\ c_1\ r_2(A)\ r_2(B)\ c_2$

[✓ Recoverable] [✓ ACR] [✓ Strict]

iii) $w_1(A)\ w_2(A)\ a_2\ c_1\ r_3(A)\ r_3(B)\ c_3$

[✓ Recoverable] [✓ ACR] ☐ Strict

Problem 3: Concurrency with Validation (12 points)

Consider a database that uses transaction validation to provide optimistic concurrency control. A user has submitted two transactions, T_1 and T_2, that have the following read and write sets:

$\text{ReadSet}(T_1) = \{A, B\}$	$\text{ReadSet}(T_2) = \{B, D\}$
$\text{WriteSet}(T_1) = \{C\}$	$\text{WriteSet}(T_2) = \{C, A\}$

According to the Validation protocol, the database will perform three steps in order for each transaction: start the transaction (*S*), validate (*V*), and finish (*F*). Please assume that the *V* step indicates the moment when both the "check" and "update VAL set" steps occur atomically, and that *F* indicates the moment when "update FIN set" occurs. Thus, when our database processes two transactions, the following 6 events will occur in some order:

T_1 *S*	T_1 *V*	T_1 *F*
T_2 *S*	T_2 *V*	T_2 *F*

In each of the following questions, write a sequence of the above 6 events to produce the given result **OR** indicate that no such schedule is possible in the box below the table.

a) (2 points) T_1 validation succeeds, but T_2 validation fails.

Time = 1	2	3	4	5	6
T1S	T2S	T1V	T2V	T1F	T2F

OR fill in this box if no such schedule is possible: ☐

b) (2 points) T_1 validation fails, but T_2 validation succeeds.

Time = 1	2	3	4	5	6
T1S	T2S	T2V	T1V	T1F	T2F

OR fill in this box if no such schedule is possible: ☐

c) (2 points) Both T_1 and T_2 validations fail.

Time = 1	2	3	4	5	6

OR fill in this box if no such schedule is possible: [✓ ☐]

d) (2 points) Both T_1 and T_2 validations succeed.

Time = 1	2	3	4	5	6
T1S	T1V	T1F	T2S	T2V	T2F

OR fill in this box if no such schedule is possible: ☐

e) (4 points) Now, assume that T1's readset is empty:

$\text{ReadSet}(T_1) = \{\}$ $\text{ReadSet}(T_2) = \{B, D\}$
$\text{WriteSet}(T_1) = \{C\}$ $\text{WriteSet}(T_2) = \{C, A\}$

Which of the following statements are true?

i) A schedule exists where T_1 validation succeeds and T_2 validation fails

[✓ True ☐] False ☐

ii) A schedule exists where T_1 validation fails and T_2 validation succeeds

[✓ True ☐] False ☐

iii) A schedule exists both T_1 and T_2 validations fail

True ☐ [✓ False ☐]

iv) A schedule exists both T_1 and T_2 validations succeed

[✓ True ☐] False ☐

Problem 4: Distributed Systems (14 points)

Consider the two-phase commit (2PC) protocol with write-ahead undo + redo logging. Assume we have a system where the only failures involve hosts halting with their disks and logs intact, followed (potentially) by reboots, with no network message loss. Suppose there is a coordinator C and two participants P1 and P2. In this question we will study the behavior of the 2PC protocol under different failure conditions.

a) (2 points) The CAP theorem applies to 2PC. Which CAP property does 2PC give up?

Availability

For parts b), c), and d), suppose we have the following sequence of events in a 2PC:

```
C sends Prepare Transaction T1 to P1,P2
P1 sends Prepared to C
P2 sends No to C
```

b) (2 points) Fill in the appropriate next message following 2PC, ignoring potential redundant retransmissions.

`__C___ sends _______Abort T1________ to _______P1,P2____________`

c) (3 points) If P1 starts **committing** T1 immediately after sending a "*Prepared*" message to C and before waiting for C's response, is the protocol still valid? Briefly explain why or why not.

Yes ☐ [✓ No]

P2 could still abort in which case P1 must abort. Thus committing is premature.

d) (3 points) If P2 starts **aborting** T1 immediately after sending a "*No*" message to C and before waiting for C's response, is the protocol still valid? Briefly explain why or why not.

[✓ Yes] No ☐

If P2 sends "No", C cannot choose to commit, so P1 must abort as well and P2 aborting is consistent.

For parts e) and f), suppose we have the following sequence of events in a 2PC:

```
C sends Prepare Transaction T2 to P1,P2
P1 sends Prepared to C
P2 sends Prepared to C
P2 crashes
P2 comes back online
C sends Commit Transaction T2 to P1, P2
P1 commits, sends DONE to C
```

e) (2 points) Assuming standard 2PC operation in the face of the above scenario, fill in what we can expect P2 to do next:

```
P2 ___commits (T2)_, sends _DONE________ to ______C_________
```

f) (2 points) P2 must make use of its log on recovery to either commit or abort. Why can P2 expect the log to be complete with respect to T2 operations in this scenario?

P2 crashed after it sent prepared. By the time it sends prepared all operations will be in the log.

Problem 5: Failure Recovery (10 Points)

a) Ben Bittwiddle is working on a fault-tolerant transaction manager using redo logging. He implemented a log that can hold Write and Commit records, and syncs to disk after each message logged. He also wrote a StorageManager that can asynchronously persist writes to stable storage (similar to Assignment 3). Unfortunately, he has observed that his implementation sometimes outputs corrupt data after a crash. In this problem, you will help Ben debug his implementation.

Ben has implemented recovery after a crash as follows, in pseudocode:

```
initAndRecover(Log L, StorageManager S):
      offset := pointer to the truncated start of L
      end := pointer to the end of L
      while (offset < end):
            R := Read record at offset in L
            if (R is a write record):
                  Queue the write that R describes to S to be persisted
                  Apply the write that R describes to the in-memory table
            Advance offset past current record
```

Assume that Ben's transaction manager is otherwise correct: writes are buffered until commit, and redo logging is used. Ben's in-memory table holds the latest committed value for all keys.

(i) **(2 Points)** In at most 3 sentences, describe Ben's error.
This implementation applies writes in the log to storage from commits that have not completed, i.e. it will apply partial commits to storage.

(ii) **(3 Points)** Describe a sequence of operations, including one crash, that Ben's code would incorrectly recover from. Describing the sequence using TransactionManager API calls from Assignment 3 is fine.
write(Txid1, Key1, foo), write(Txid1, Key2, bar),
commit(Txid1) <- crash during, after appending the first write to the log.

b) Ben's friend Alyssa P. Hacker has implemented a similar transaction manager, also using redo logging, but she is also encountering data corruption after recovery. She implemented a function writePersisted that is called whenever writes to the StorageManager make it to disk as follows:

```
writePersisted(long key, long log_offset_of_persisted_write, byte[] value):
      old_truncation := pointer to the truncated start of L
```

```
if (old_truncation < log_offset_of_persisted_write):
      Truncate L up to log_offset_of_persisted_write
```

In the above pseudocode, L is the log. The function writePersisted is called whenever the StorageManager persists a write that was queued to it earlier. The StorageManager guarantees that writes to the same key are always persisted in order, but writes to different keys may be persisted out of order. You can assume that Alyssa's redo logging implementation is correct outside of the writePersisted method.

(i) **(2 Points)** In at most 3 sentences, describe Alyssa's error.

When writes are persisted out of order, this implementation truncates the log past unpersisted writes, leading to possible inconsistency after recovery.

(ii) **(3 Points)** Describe a sequence of operations, including one crash, that Alyssa's code would incorrectly recover from. Describing the sequence using TransactionManager API calls from Assignment 3 is fine.

write(Txid1, key1, foo), write(Txid1, key2, bar), commit(Txid1), writePersisted(key2, some offset, bar), crash.

Problem 6: Cost Modeling of Different Storage Formats (15 points)

In this problem, we'll use cost models to explore the best in-memory storage format for various queries (similar to Assignment 1). Consider a database with N rows and C columns (c_0, c_1, …, c_{C-1}), all stored as 4-byte integer fields. Each field is drawn uniformly at random from the range [0, K). Recall that in a row-oriented storage format, values for all columns of row 0 are stored in contiguous memory locations, then all columns of row 1, and so on. In a column-oriented storage format, values for all rows of column 0 are stored in contiguous locations, then all rows of column 1, and so on. Assume that $N \gg 16$, $C \gg 16$, N mod 16 = 0, C mod 16 = 0, and $N \gg K$.

You can assume that every time a memory address not in cache is accessed (read or write), the corresponding cache line is loaded into cache. A cache line is 64 contiguous bytes in memory. **Assume that loading one cache line from memory into cache has a cost of 1**. The system has a **cache size of 16 cache lines** and the least recently accessed cache line is evicted back to memory when the cache is full. **Assume that accesses to cache have no cost and also that computation has no cost**.

Consider the following queries. Each of them is executed with an **outer loop over the rows** (i.e. work for each row is completed before moving to the next one) and **inner loop over the columns from left-to-right**. Compute the **expected cost of each one in terms of N, C, and/or K** for each storage format. For example, SUM(c_0) has a row-oriented cost of N and a column-oriented cost of N/16.

a) (3 points) SUM($c_0 + c_1 + … + c_7$)

i) Row-oriented: N
ii) Column-oriented: N/2

c_0, c_1, …, c_7 for a given row (= 8*4 = 32 bytes) fit in a cache line for the row-oriented store, so N cache lines accessed in total. Each column takes N/16 cache lines for the column-oriented store, so total number of cache lines for 8 columns is N/2.

b) (3 points) SUM($c_0 + c_1 + … + c_{15}$)

i) Row-oriented: N
ii) Column-oriented: N

c_0, c_1, …, c_{15} for a given row fit in a cache line (64 bytes) for the row-oriented store, so as before, N cache lines accessed in total. Each column, as before, takes N/16 cache lines, giving N cache lines in total for the column-oriented store as well.

c) (3 points) SUM($c_0 + c_1 + \ldots + c_{C-1}$)

i) Row-oriented: NC/16

ii) Column-oriented: NC

To access each row for the row-oriented store, we need C/16 cache lines. Thus for N rows, we need to load NC/16 cache lines. For the column-oriented store, cache lines for a column c_i no longer remain in cache when accessing values for the next row (remember that column values for each row are accessed left-to-right, and that processing for a row is completed before moving onto the next row). Thus, the number of cache lines that need to be read for every row is C, so total number of cache lines for N rows is NC.

The above queries did not have predicates. However, predicates are common, so we'd like our database to support them. For queries with predicates, we also consider two more data structures:

- An index that maps each distinct value in column c_{C-1} to a list of rows that have that value in c_{C-1}. **The cost of retrieving the row lists for M contiguous values of c_{C-1} from this index is dM for some small constant d**. Assume that reading through the row lists themselves has no cost. This index is used along with row-oriented storage.
- A row store that is sorted in ascending order of c_{C-1}.

Compute the **expected cost of the following query in terms of N, C, K, d, and/or T** for each storage format.

d) (6 points) SUM($c_0 + c_1 + \ldots + c_{C-1}$) WHERE $c_{C-1} < T$

i) Row-oriented, unsorted: N + (T/K)N(C/16)

ii) Column-oriented, unsorted: (N/16) + (T/K)NC

iii) Indexed with unsorted row-oriented storage: dT + (T/K)N(C/16)

iv) Row-oriented, sorted on c_{C-1}: (T/K)N + (T/K)N(C/16)

Each cost has two sub-costs: one to determine the rows satisfying the given predicate, and another to compute sums for satisfying rows. The expected cost to compute the sum for all formats can be computed similar to part c). For the unsorted formats, all values in c_{C-1} need to be read (which is N for the row-oriented format, N/16 for the column-oriented format). For the sorted format, only values up to the threshold T need to be read, which takes (T/K)N cache lines. With the index, only rows with c_{C-1} in the range [0, T) need to be looked up (dT cost).

Problem 7: Differential Privacy (12 points)

Recall that the *sensitivity* of a function f on sets is defined as $\Delta f = \max |f(D_1) - f(D_2)|$, where the max is over all pairs of sets D_1, D_2 that differ in at most one element.

SecretBuddies is a privacy-conscious social network whose data consists of a set called Users that has a record with the ID, name, age, gender, address, likes and friend list of each user. (Friendships on SecretBuddies are 2-way: if user A is friends with B, then B is friends with A). Moreover, the users' ages are all between 0 and 150, and each user on SecretBuddies is only allowed to have up to 100 friends (can you really have more secret buddies?).

SecretBuddies wants to allow its data scientists to make some queries on Users, but is worried about the differential privacy of each query. For each of the following queries on Users, compute its sensitivity, or write "unbounded" if the sensitivity is not bounded by a constant we can compute using the information in the question. *(Please justify your answers. Assume that if a user is removed from the network, that user also no longer appears in any friend lists.)*

a) (2 points) f_1(Users) = # of users who are male, live in Palo Alto, and like My Little Pony

1: Adding or removing a user can only change this count by 0 or by 1.

b) (2 points) f_2(Users) = maximum age of users who like My Little Pony

150: The biggest change possible would be if we added a user with age 150 to a dataset where every other user has age 0.

c) (2 points) f_3(Users) = # of users named "Alice" who are friends with someone named "Bob"

100: The biggest change possible would be if we remove a user named "Bob" who had 100 friends, all named "Alice".

d) (3 points) f_4(Users) = # of users connected, through some path in the friendship graph, to Zack Morsey (the CEO of SecretBuddies)

Unbounded: There might be a user, such as a friend of Zack Morsey's, that is on a path from Zack to every other user in the graph, so we can't bound the number of paths broken by removing this user using a constant.

e) (3 points) f_5(Users) = # of triangles in the friendship graph (a triangle is a set of three distinct users {A, B, C} that are all friends with each other)

4950: Each individual can be part of at most (100 choose 2) = 100 * 99 / 2 = 4950 triangles, because the user can have up to 100 distinct friends, and if those are all friends with each other, that would create 100 choose 2 distinct triangles.