CS 245 Final Exam
Winter 2022 Solutions

- Please read all instructions carefully. In case of any ambiguity, write any assumptions you made in your answer. We can't answer questions from students live during the exam, but tell us about anything you found unclear later (in a private Ed post).

- There are six problems for a total of 85 points. You have 3 hours to **take and upload** the exam to Gradescope (**please give yourself ~5 minutes to upload within the 3 hours)**. If you have issues uploading the test on Gradescope, please upload it on this Google Form, but please beware that we may penalize late submissions.

- Unless you are taking the exam in person, you need to upload your solutions as a PDF that **exactly matches the page numbering** of this document (the same questions are on the same pages). You can do this by printing and scanning the exam, by annotating the answers on the PDF for the exam, or by downloading the exam here as a Word document or as a copy on Google Docs, filling it in and saving that as a PDF (make sure that each question **stays on the right page** though). Ensure that the file you upload to Gradescope is sharp, legible, and aligned.

- The test is open-book, but you are not allowed to communicate with other people to do it. This includes asking public questions on Ed. You may also use the Internet during the test, but keep in mind that many online resources might use terms differently from our course, and that directly copying an online resource is considered plagiarism and is not allowed. We don't think you will need resources other than the course materials.

- Solutions will be graded on correctness and clarity. For the long-answer problems, please show intermediate work. Each problem has a relatively simple to explain solution, and we may deduct points for overly complex answers. Partial solutions will get partial credit.

NAME: ____

SUID: ____

In accordance with both the letter and spirit of the Stanford Honor Code, I have neither given nor received assistance on this test. A typed signature is fine. If uploading answers as a separate document, please include your Name, SUID, and Signature clearly on your submission.

SIGNATURE: ____

Problem 1: Short Answers (19 points)

a) (2 points) Which of the following are true? *(Circle or highlight all correct answers.)*

(a) Columns tend to compress better than rows due to lower information entropy.
(b) It is typically faster to modify a record in a column store than a row store.
(c) Data warehouse workloads tend to involve large scans of fewer columns, making column stores more suitable than row stores for data warehouses.
(d) Performance can increase if queries are executed directly on compressed data.

b) (2 points) Suppose that we have a table called Painting with attributes artist, year, and style. Which of the following are valid rewrites of the query $\Pi_{\text{artist}}(\sigma_{\text{style = 'impressionism'}}(\text{Painting}))$? *(Circle or highlight all correct answers.)*

(a) $\Pi_{\text{artist}}(\sigma_{\text{style = 'impressionism'}}(\Pi_{\text{artist}}(\text{Painting})))$
(b) $\Pi_{\text{artist}}(\sigma_{\text{style = 'impressionism'}}(\Pi_{\text{style}}(\text{Painting})))$
(c) $\sigma_{\text{style = 'impressionism'}}(\Pi_{\text{artist}}(\text{Painting}))$
(d) $\Pi_{\text{artist}}(\sigma_{\text{style = 'impressionism'}}(\Pi_{\text{artist, style}}(\text{Painting})))$

c) (2 points) Consider an extendible hash table that uses the first i bits of b bits output by a hash function to map it to a bucket. Suppose that each bucket can hold at most 2 keys and that the table starts empty. We then insert keys with the following hashes, in the following order:

01100, 10010, 00110, 11111, 01010, 01110

After the insertions, the global depth is __3__, and the local depth of the bucket containing 00110 is __2__.

d) (2 points) Undo-redo logging often leads to better performance than either undo logging or redo logging alone. What is the primary reason for this? *(Circle or highlight one answer.)*

(a) Undo-redo logging requires less disk space for the log.
(b) Undo-redo logging requires fewer disk I/Os than either of the other schemes before a transaction can commit.
[✓ (c) Undo-redo logging gives the database more flexibility to decide when to write in-memory pages to disk.]
(d) Undo-redo logging requires fewer CPU cycles during logging.

e) (2 points) Consider the following undo-redo log with checkpointing that crashes at line 8.

Which changes will be un-done (list the line number(s))? __5__
Which changes will be re-done (list the line number(s))? __4__

0	<START, txn1>
1	<WRITE, txn1, x, 5, 6>
2	<START, txn2>
3	<START CHECKPOINT>
4	<WRITE, txn1, z, 7, 4>
5	<WRITE, txn2, y, 9, 10>
6	<END CHECKPOINT>
7	<COMMIT, txn1>
8	CRASH

f) (3 points) The following transaction schedules use the notation in class (for example, $w_1(A)$ means that transaction 1 writes object A). For each schedule, specify whether it is recoverable and whether it avoids conflicting rollback (ACR):

Schedule	Recoverable?	ACR?
$w_1(A)\ r_2(A)\ c_1\ w_2(A)\ c_2\ r_3(A)\ a_3$	✓	
$w_1(A)\ w_2(A)\ c_2\ a_1\ r_3(A)\ c_3$	✓	✓
$w_1(A)\ r_2(A)\ c_2\ c_1\ r_3(A)\ c_3$		

g) (2 points) Suppose we have a Dynamo cluster with three nodes (A, B, C), using ⅔ quorums. Consider a key x, which originally has the value 7 on all nodes. A client makes a put to change the value to 8. Assume the put operation is successful. For another client later performing reads to x, which of the following results are possible (A: 8 means the client asked node A to read the value and received 8 back)? *(Circle or highlight all correct answers.)*

(a) A: 8, B: 7, C: 7
[✓ (b) A: 8, B: 8, C: 8]
[✓ (c) A: 8, B: 7, C: 8]
[✓ (d) A: 7, B: 8, B: 8]

We also accepted just b and c given the typo in (d) which should say A: 7 B: 8 C: 8.

h) (2 points) CockroachDB used range partitioning instead of hash partitioning for its key-value storage layer. What is one advantage of this choice? *(Circle or highlight one answer.)*

(a) Range partitioning reduces the amount of work required to figure out which nodes a given key is on.
(b) Range partitioning reduces the amount of storage required for metadata about a table.
[✓ (c) Range partitioning reduces the cost of queries that involve lexicographically nearby keys.]
(d) Range partitioning can support stronger transaction isolation levels.

i) (2 points) Suppose that we are implementing a differentially private query system that stores a dataset about patients' in a hospital, and we know that each patient's age is between 0 and m. What is the sensitivity of each of the following queries we might run on the dataset (according to the definition of sensitivity in class)?

Query	Sensitivity
`SELECT COUNT(*) FROM Patient`	1
`SELECT COUNT(*) FROM Patient WHERE disease != "flu"`	1
`SELECT MAX(age) FROM Patient`	m
`SELECT MIN(age) FROM Patient WHERE disease = "flu"`	m

Problem 2: Optimistic Concurrency (12 points)

a) (6 points) Consider two separate transaction schedules with numbered operation sequences in the tables below. R(X) means read X and W(X) means write X. The database uses validation (optimistic concurrency) for concurrency control. ***Note:*** *Ignore the "Finish" events in the table for transactions that you don't think will validate.*

Circle or highlight the answer of the following questions. For transactions that abort, fill in the operation number they abort on.

Operation	1	2	3	4	5	6	7	8
T1			R(C)	W(D)		Validate		Finish
T2	W(C)	R(D)			Validate		Finish	

T1 will A) Succeed [✓ B) Abort (Operation __6__)]

T2 will [✓ A) Succeed] B) Abort (Operation _____)

Operation	1	2	3	4	5	6	7	8
T3	R(A)	W(B)			Validate	Finish		
T4			W(B)	R(A)			Validate	Finish

T3 will [✓ A) Succeed] B) Abort (Operation _____)

T4 will [✓ A) Succeed] B) Abort (Operation _____)

b) (3 points) Which of the following sequences of operations can be achieved by validation? *(Circle or highlight all correct answers.)*

[✓ (a) $W_1(A)$, $W_1(B)$, $W_3(A)$, $W_2(A)$]
(b) $W_1(A)$, $R_2(A)$, $W_1(C)$
[✓ (c) $W_1(A)$, $W_2(B)$, $W_1(C)$]

c) (1 points) If a sequence of operations can be achieved by validation, then it must also be achievable with 2-phase locking. *(Circle or highlight true or false)*

[✓ A) True] B) False

d) (2 points) Fill in the blank with two situations where concurrency control with 2-phase locking is likely to perform better than validation.

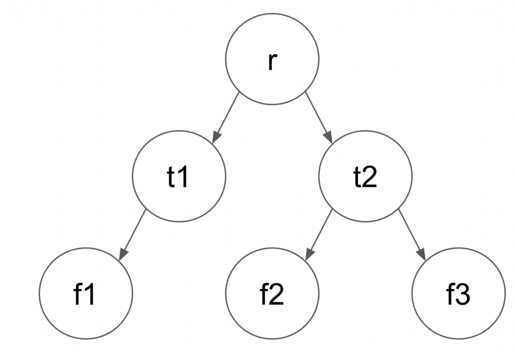
Situation 1: ______________________________________

Situation 2: ______________________________________

Conflicts are frequent; system resources are lacking; have loose latency constraints

Problem 3: Locking (14 points)

a) (8 points) Consider the schedule of operations below on a relation r that contains multiple tuples (t1 and t2) and fields in each tuple (f1, f2, f3). We run three different transactions, which each submit various instructions over time, represented as “Ix”. These transactions will also acquire hierarchical locks using the scheme in class (S, X, IS, IX, SIX), using strict 2PL, before running their instructions. Can you find the **order** in which the instructions in these transactions will be executed? Note that the execution of this schedule may not match the submission order exactly due to waiting for locks. Assume that a transaction commits right after its last operation.



Timestamp	0	1	2	3	4	5	6	7	8
T1	I0: R(f1)				I4: W(f2)			I7: W(f3)	I8: R(r)
T2		I1: R(t1)	I2: R(t2)				I6: W(f3)		
T3				I3: W(f2)		I5: W(t1)			

Fill out the order the instructions will run during execution and list out all the locks each instruction newly acquires using hierarchical locking for the transaction. No need to list the locks the transaction already has. If there are multiple valid orders, fill any of the valid orders. Timestamp 0 is given to you. (You do not need to fill in all the blanks for Lock(s) rows)

Timestamp	0	1	2	3	4	5	6	7	8
Instruction	I0	I1	I2	I6	I4	I7	I8	I3	I5
Lock(s)	IS(r)	IS(r)	S(t2)	IX(r)	IX(r)	X(f3)	SIX(r)	IX(r)	X(t1)
	IS(t1)	S(t1)		SIX(t2)	IX(t2)			IX(t2)	
	S(f1)			X(f3)	X(f2)			X(f2)	

b) (4 points) Fill out an order of instructions that will result in a deadlock during execution. You do not need to fill in all the blanks, just until the deadlock occurs. Explain why the deadlock occurs, and one way to resolve the deadlock after it occurs.

Timestamp	0	1	2	3	4	5	6	7	8
Instruction	I0	I1	I2	I6	I3				

The deadlock occurs because T3 holds the X lock on f2 but is waiting on the X lock on t1, whereas T1 holds the IX lock on t1 but is waiting on the X lock on f2. Since we are following a strict 2PL schedule, neither transaction will release its lock until after acquiring all the necessary locks. Thus, the deadlock occurs.

One way to resolve the deadlock after it occurs is to abort and restart one transaction.

c) (2 points) You have a table called `inventory` containing all your inventory items and their individual item price, and you decide to increase the price(s) of the cheapest item(s) by 1 dollar. To perform such a transaction, which hierarchical lock should this transaction place on `inventory` and why? Assume there is no previous lock on the table.

This transaction should place the SIX lock on `inventory`. We have S because we need to read all of the item prices to determine which is the minimum during execution. We have IX because we intend to increment the lowest price(s) by 1. (We will use X to lock the lowest prices)

Note that IX does not work because you do not want other transactions to have the same IX lock on `inventory` and modify prices before you complete your modification. And X is not the best solution because you are preventing other transactions from reading/modifying other parts of `inventory`.

Problem 4: Logging and Recovery (16 Points)

You will be filling in code for an atomic and durable transaction manager class, similar to assignment 3, but based on *undo-logging* (assignment 3 used redo logging). The transaction manager has the variables: LogManager lm, StorageManager sm, HashMap<long, byte[]> map, and HashMap<long, List<long> activeTransactions. They have the APIs below **(note that the StorageManager's put API is synchronous, unlike the assignment; there's no queuing):**

LogManager:

- `appendLogRecord(Record)`
- `Record readLogRecord(int offset, int size)`

StorageManager:

- **`persistKeyValueToDisk(long key, byte[] value)`** *`// returns after write succeeds`*
- `HashMap<long, byte[]> readPersistedMap()`
- `byte[] readValue(long key)`

In memory map (used to service reads):

- `map.put(long key, byte[] value)`
- `byte[] map.get(long key)`

Record has the following constructor (key and value are ignored unless it is a WRITE record)

- `Record(int type, txID, long key, byte[] value)`
- Available types: WRITE, COMMIT, ABORT, START

Assume that start(), shown below as an example, and abort() are already implemented; abort adds an abort record to the log and removes the txID from active transactions. *Assume in any transaction, a key is written to at most once.*

```
public void start(long txID) {
      lm.appendLogRecord(new Record(START, txID));
      if activeTransactions.containsKey(txID):
            activeTransactions.put(txID, new ArrayList<>());
}
```

a) (4 points) Use the member variables and function APIs above to fill in the blanks:

```
public void write(long txID, long key, byte[] value){
  lm.appendLogRecord(new Record(WRITE, txID, key, map.get(key) OR
sm.readValue(key)));


  sm.persistKeyValueToDisk(key, value);
  activeTransactions.get(txID).add(key);
}
```

b) (4 points) Use the member variables, function APIs and parameters to fill in the blanks:

```
public void commit(long txID) {
  for (long key: activeTransactions.get(txID)) {

   map.set(key, sm.readValue(key));
  }

  lm.appendLogRecord(new Record(COMMIT, txID));
  activeTransactions.remove(txID);
}
```

Note: given this code would mess up atomicity in the case of transactions that call writes and commit concurrently, we gave everyone points for this question.

c) (8 points) For blanks that contain [true | false], choose between true or false. For empty blanks, write in the correct code.

```
public void initAndRecover(StorageManager sm, logManager lm) {
  this.sm = sm; this.lm = lm; this.map = sm.readPersistedMap();
  ArrayList<Record> records = loadRecords(lm);
  HashSet<long> committedIDs = scanCommittedIDs(records);
  HashSet<long> abortedIDs = scanAbortedIDs(records);

  for (Record record: Collections.reverse(records)) {
    if record.ty == WRITE and !committedIDs.containsKey(record.txID) {

      this.map.set(key,record.value));

      this.sm.persistKeyValueToDisk(key, record.value);
    }
  }
  for (Record record: records) {
    if (committedIDs.containsKey(record.txID) == [true | false]) {
      if (abortedIDs.containsKey(record.txID) == [true | false]) {

       lm.appendLogRecord(new Record(ABORT, txID));
      }
    }
  }
}
```

Problem 5: Distributed Transactions (12 Points)

a) (4 points) Which of the following statements are **true** about the 2PC protocol discussed in class? *(Circle or highlight all correct answers.)*

[✓ (a) 2PC guarantees atomic commits.]
[✓ (b) 2PC adheres to CAP theorem because it can run into conditions where a node has frozen new transactions and is waiting on a COMMIT message from the coordinator.]
(c) In case that a single participant and the coordinator crash, the remaining nodes who had sent a PREPARED to the coordinator can conduct a majority vote to elect a coordinator who tells them to commit / abort the transaction.
[✓ (d) 2PC requires nodes to remember some of their past actions in durable storage.]

Next, let's think carefully about each step of the 2PC protocol:

- Step 1: Coordinator asks participants to PREPARE.
- Step 2: Participants respond that they are either PREPARED or NOT PREPARED.
- Step 3: Coordinator asks participants to COMMIT/ABORT based on the votes it got.

b) (2 points) Suppose each participant responds to the coordinator by first sending its reply and then logging whether it prepared to durable storage. State why this can be problematic.

Should log first and then send the message. This will ensure that even if the participant crashes and comes back up, it can assume that it was supposed to be prepared and not change its mind about its own state in case it sent out any messages to the coordinator already.

(c) (2 points) State a problem which can arise if the coordinator always waits until **all** participants have responded with a PREPARED/NOT PREPARED.

The participants will not be able to perform any other transactions since they are PREPARED but some other node could have crashed, causing the coordinator to wait indefinitely until all participants respond.

(d) (2 points) Let's say the coordinator asked all nodes to COMMIT. However at the time, one participant node was down and could not receive the COMMIT message. Upon coming back up it sees that since it was PREPARED before crashing, it commits anyway. In this particular case, this didn't cause an issue. However, explain when and why this approach can be problematic.

In case the coordinator had aborted the transaction instead, this would have violated the atomicity of the transaction.

e) (2 points) There is a famous distributed systems problem called the Generals' Paradox, where two Generals (G1, G2) are out on a campaign. It is night time, and in the morning, they need to march together on an objective that they want to capture. Either General might choose not to march upon assessing the conditions of their camp and battalion strength. If they march **together,** they are assured of success, but if any General marches alone, they will be **defeated**.

To decide what to do, the Generals can choose to send multiple messengers over the night to coordinate whether they are planning to march in the morning, but there is no guarantee that the messengers will make it to the other General.

For example, the exchange might look like this:

```
G1  ------- I am not planning to march tomorrow (lost) ----> G2
G1  <------ I am planning to march tomorrow (received) ----- G2
G1  <------ I did not receive your message yet (lost) ------ G2
```

After this exchange, G2 does not know whether G1 plans to march in the morning, but it is also hard for G2 to decide not to march, because G1 might've received their message and plan to go.

One can observe that this Generals' Paradox problem is very similar to the atomic commitment problem handled by 2PC. However, the Generals' Paradox is known to have **no solution**, as it can run into failure scenarios as described above.

Describe in a few sentences what makes this problem different from 2PC.

2PC is allowed to take arbitrarily long, unlike this problem, where the Generals need to coordinate within a certain window of time. 2PC also differs in that it centralizes the decision to commit in a single place, i.e., the commit coordinator. 2PC needs nodes to coordinate to come to a decision *eventually* and not at the same time.

Problem 6: Streaming (12 points)

Dapple is an up-and-coming music app that wants to use a streaming SQL database, based on Stanford CQL, to analyze how its users listen to music. The database stores two tables:

- User, with fields userId and plan ("free" or "premium").
- Play, a table that records when users play songs, with fields userId, song, and eventTime.

Users can sign up for the service, delete their account, or switch between free and premium plans on Dapple's website, which results in transactions that change the User table. They can also play songs on their phone, which uploads a record in the Play table. These play events may take a while to reach the database if the phone is offline (Dapple lets users play songs offline).

Using the CQL syntax in our slides (ISTREAM, DSTREAM, RSTREAM, etc), write how Dapple should express the following stream queries. Each query should return a **relation**, not a stream. We've written the first one for you as an example.

Note: The ISTREAM, DSTREAM and RSTREAM operators extend the records in each table with a new field, t, to represent processing times. Assume this field is just called "t" in SQL.

a) (0 points) Count the number of users who signed up in the past hour.

SELECT COUNT(*) FROM ISTREAM(User) [RANGE 1 HOUR]

b) (2 points) Count the number of users who deleted their account in the past hour.

SELECT COUNT(*) FROM DSTREAM(SELECT userId FROM User) [RANGE 1 HOUR]

Note: This was trickier than expected because we had a mistake in our example above; we also gave full points for FROM DSTREAM(User) here, but it's incorrect because it would show users who changed plans during the past hour. Our part a) example should have used ISTREAM(SELECT userId FROM User) like this answer.

c) (2 points) Count the number of play events that were received at least 1 hour after they happened on the user's phone (assuming their eventTime and the database's clock are accurate).

SELECT COUNT(*) FROM ISTREAM(Play) [RANGE UNBOUNDED]
WHERE (t - eventTime) >= 1 HOUR

Note: It's OK to omit [RANGE UNBOUNDED] as that is the default in CQL.

d) (2 points) Count the total number of plays of the song “Baby Shark” in 2022.

```
SELECT COUNT(*) FROM Play
WHERE song = “Baby Shark” AND YEAR(eventTime) = 2022
```

Note: could also use ISTREAM(Play).

e) (3 points) Count the users who listened to “Baby Shark” at a time when they were on the “premium” pricing plan.

```
SELECT COUNT(DISTINCT userId)
FROM Play AS p
  JOIN RSTREAM(User) [RANGE UNBOUNDED] AS u
    ON p.userId = u.userId AND p.eventTime = u.t
WHERE song = “Baby Shark” AND u.plan = “premium”
```

f) (3 points) Return the IDs of all the users who upgraded to the premium plan sometime after the first time they listened to “Baby Shark”.

```
SELECT DISTINCT userId
FROM Play AS p
  JOIN ISTREAM(User) [RANGE UNBOUNDED] AS u
    ON p.userId = u.userId
WHERE u.plan = “premium” AND u.t > p.eventTime
```