6.814/6.830 Database Systems: Fall 2017 Quiz II

There are 13 questions and 17 pages in this quiz booklet. To receive credit for a question, answer it according to the instructions given. You can receive partial credit on questions. You have 80 minutes to answer the questions.

Write your name on this cover sheet AND at the bottom of each page of this booklet.

Some questions may be harder than others. Attack them in the order that allows you to make the most progress. If you find a question ambiguous, be sure to write down any assumptions you make. Be neat. If we can’t understand your answer, we can’t give you credit!

THIS IS AN OPEN BOOK, OPEN NOTES QUIZ.
YOU MAY USE A LAPTOP OR CALCULATOR.
YOU MAY NOT ACCESS THE INTERNET.

Do not write in the boxes below

<table>
<thead>
<tr>
<th>1-3 (xx/24)</th>
<th>4-7 (xx/34)</th>
<th>8-10 (xx/18)</th>
<th>11-13 (xx/24)</th>
<th>Total (xx/100)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Name:
I Dynamo

Consider the Dynamo configuration shown in Figure 1. Here, N=2, W=2, and R=1, and the system is using sloppy quorums. The network is partitioned, such that nodes A, B, and C can talk to each other but not nodes D, E, and F, and vice-versa. Assume the ring is ordered clockwise (e.g., B is A’s successor). Further, assume we are using the “load balancer” configuration as described in Section 4.5 of the Dynamo paper, such that “the node that receives [a] request will not coordinate it if the node is not in the top N of the requested key’s preference list”.

1. [6 points]: Suppose node D receives a write to key $k_1$ while the network is partitioned. Which nodes record this write?

(Circle all that apply.)

A. Node A  
B. Node B  
C. Node C  
D. Node D  
E. Node E  
F. Node F  

Answer. 
Node D and Node F

Name:
Node F is the first node in the preference list for $k_1$. Because we have sloppy quorum, we proceed clockwise until we get to a reachable node, which is Node F.
2. [6 points]: Suppose the partition heals and node D receives a write to key $k_1$. Which nodes record this write?

(Circle all that apply.)

A. Node A
B. Node B
C. Node C
D. Node D
E. Node E
F. Node F

Answer. **Node A and Node F**
D forwards the request to the first node in the preference list, F. Then we proceed to the next node in the preference list which is A.
II Two-phase commit

Suppose you run the following transactions on a two-node database system, with one coordinator \( C \) and one subordinate \( S \) using two-phase commit. These transactions use rigorous two-phase locking, and are all coordinated by the same node. The statements below do not indicate a particular interleaving, but just the reads and writes each transaction performs.

<table>
<thead>
<tr>
<th></th>
<th>T1</th>
<th>T2</th>
<th>T3</th>
</tr>
</thead>
<tbody>
<tr>
<td>RA</td>
<td>RA</td>
<td>RB</td>
<td></td>
</tr>
<tr>
<td>WA</td>
<td>WA</td>
<td>WB</td>
<td></td>
</tr>
<tr>
<td>RC</td>
<td>RD</td>
<td>RE</td>
<td></td>
</tr>
<tr>
<td>WC</td>
<td>WD</td>
<td>WE</td>
<td></td>
</tr>
</tbody>
</table>

You are told the following facts about the behavior of the two-phase commit protocol:

- T1 began the commit protocol before T2
- C sent S a prepare message for T1
- S wrote a prepare record for T2
- S sent C a “Yes” vote for T3
- S sent C an “Ack” for T2

3. [12 points]:

For each of the outcomes listed below, indicate whether it is consistent with the above facts assuming the system is running the presumed abort variant of the protocol. Here “pending” means that the transaction has not yet committed or aborted.

(Circle “Yes” if the outcome is consistent, “No” otherwise)

A. Yes / No  T1: Commit; T2: Commit; T3: Abort
B. Yes / No  T1: Pending; T2: Abort; T3: Commit
C. Yes / No  T1: Abort; T2: Abort; T3: Commit
D. Yes / No  T1: Commit; T2: Pending; T3: Pending
E. Yes / No  T1: Commit; T2: Commit; T3: Pending
F. Yes / No  T1: Pending; T2: Commit; T3: Pending

Name:
Answer.
T2 committed since coordinator must have received COMMIT in order to send ACK.

A. Yes
B. No T1 can’t be pending. T2 Committed
C. No T2 Committed
D. No T2 Committed
E. Yes
F. No T1 can’t be pending if T2 committed since T1 began the commit protocol before T2.
III OCC

Consider three concurrent transactions, A, B, and C, running OCC, reading and write three records, X, Y, and Z, with the following read and write sets:

<table>
<thead>
<tr>
<th>Transaction</th>
<th>Read</th>
<th>Write</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>{X}</td>
<td>{X}</td>
</tr>
<tr>
<td>B</td>
<td>{X}</td>
<td>{Y}</td>
</tr>
<tr>
<td>C</td>
<td>{Y}</td>
<td>{Z}</td>
</tr>
</tbody>
</table>

Suppose the transactions all complete their read phase and enter the validation phase in A, B, C order. Assume they run the serial validation protocol described in the paper by Kung and Robinson, and that Transaction A is the first transaction run by the system.

4. [6 points]: For each transaction, indicate its outcome:
   (Circle COMMIT or ABORT for each item.)

A. COMMIT ABORT
B. COMMIT ABORT
C. COMMIT ABORT

Answer. Commit. No transactions before.
Abort. Read X conflicts with the previous transaction write X.
Commit. Checks with Transaction A which only wrote X (B aborted) and commits due to no conflicts.
IV  BigTable

5. [10 points]:
Which of the following statements about BigTable are true:

(Circle True or False for each item below.)

A. True / False  If Chubby fails, BigTable will become unavailable

B. True / False  If the master fails, BigTable will become unavailable

C. True / False  Data is physically partitioned by key range and column-family

D. True / False  Two SSTables in the same Tablet may contain overlapping ranges of keys

E. True / False  BigTable depends on compactions to ensure deleted results are never visible to clients

Answer.
A. T - BigTable is dependent on the reliability of Chubby for a number of tasks and will therefore be unavailable if Chubby fails (see end of section 4 in BigTable paper)
B. F - If the master fails, a new worker will become master and BigTable will continue to run
C. T and F - The answer was originally intended to be true as data can be organized into locality groups based on column family, but because it’s not necessary that data is organized by locality groups, we gave credit for both answers
D. T - SSTables are written periodically and not partitioned by key
E. F - BigTable uses compactions to save space, but the compactions are not necessary to prevent the clients from seeing deleted results

Name:
V  Distributed Query Processing

Consider a distributed database with 3 tables A, B, and C. Here, B.c and C.e are primary keys of B and C, respectively. Users of this database run only the following query:

```sql
SELECT A.a, SUM(A.b), SUM(B.b), SUM(C.c)
FROM A, B, C
WHERE A.a = B.c AND B.d = C.e
GROUP BY A.a
```

The distributed database runs on a cluster of 20 machines with 100GB of RAM and 500 GB of disk each.

6. [10 points]: If all the tables (A,B,C) are 1 TB and have about the same number of records, what do you think is the best way to partition the data and execute the query? Assume that the initial partitioning of data has already been done (i.e., does not incur any additional cost.) Provide the best initial data partitioning layout and the query plan. In your plan indicate the type of join and also where shuffle operations need to be performed. It is sufficient to show the plan that will run on only one node, assuming each node runs the same plan (see Figure 2 for an example showing a join between two tables that need to be shuffled.)

(Write your answer in the space below.)

**Answer.** Hash partitioning A on A.a, B on B.d, C on C.e. This is better than joining A and B first because the output of the final join is in A.a order which is helpful for the group by.
7. [8 points]: Instead, if table A and B are of size 1 TB with about the same number of records and table C is of size 50 GB with many fewer records, what do you think is the best way to partition the data and execute the query? Provide the best initial data partitioning layout and the query plan.

(Write your answer in the space below.)

Answer. Hash partitioning A on A.a, B on B.d. Replicate C on all machines. This avoids shuffling the output of the join as it is produced in A.a order.

![Query Plan Diagram]

Figure 4: Query Plan.
VI Serializability and Two Phase Locking

Below, we provide 3 interleavings of several concurrent transactions (consisting of READ and WRITE statements on objects).

Your job is to indicate whether, for each of the interleavings:

- The interleaving has an equivalent serial ordering.
- Whether the interleaving would be permitted/could arise under strict two-phase locking (where write locks are held until end of transaction).
- Whether the interleaving would be permitted/could arise under rigorous two-phase locking (where read and write locks are held until end of transaction).
- Whether the ordering would be valid using snapshot isolation. Assume that snapshot isolation is implemented in the same way as optimistic concurrency control in the paper by Kung and Robinson, except that read sets are not tracked (and read-write / write-read conflicts are ignored.) You can assume that serial validation is used the write phases of optimistic concurrency control happen during the COMMIT statement (and no sooner.)

Assume in all cases that written values can depend on previously read values. For locking protocols, assume that locks are acquired just before a the data item they cover is read or written, and they are released as soon as the locking protocol permits.

Name:
8. [6 points]:

1. T1: READ A
2. T2: READ A
3. T2: WRITE B
4. T1: WRITE A
5. T1: READ C
6. T1: COMMIT
7. T2: COMMIT

Circle ‘true’ or ‘false’ for each choice.

A. True / False  The interleaving is serializable
B. True / False  The interleaving is permitted under strict two-phase locking
C. True / False  The interleaving is permitted under rigorous two-phase locking
D. True / False  The ordering is valid using snapshot isolation

Answer.
A. T - The dependency graph has just one arrow from READ A in T2 to WRITE A in T1, so no cycle
B. T - T2 can release its S lock on A after acquiring the lock on B, so T1 can acquire its X lock on A. No other locks conflict
C. F - T1 cannot write A because T2 has not yet released its S lock on A
D. T - T1 and T2 have disjoint write sets

Name:
9. [6 points]:

1. T1: READ A
2. T3: READ A
3. T2: WRITE B
4. T2: COMMIT
5. T3: WRITE C
6. T1: WRITE B
7. T1: COMMIT
8. T3: COMMIT

Circle ‘true’ or ‘false’ for each choice.

A. True / False  The interleaving is serializable
B. True / False  The interleaving is permitted under strict two-phase locking
C. True / False  The interleaving is permitted under rigorous two-phase locking
D. True / False  The ordering is valid using snapshot isolation

Answer:
A. T - The dependency graph has just one arrow from WRITE B in T2 to WRITE B in T1, so no cycle
B. T - T2 can release its X lock on B after acquiring the lock on B, so T1 can acquire its X lock on B. No other locks conflict
C. T - T2 can release its X lock on B after committing, so T1 can acquire its X lock on B. No other locks conflict
D. F - T2 writes B in between the start and end of T1, so T1 would abort because B has changed
10. [6 points]:

1  T1: READ B
2  T2: READ A
3  T2: WRITE B
4  T2: COMMIT
5  T1: WRITE A
6  T1: COMMIT

Circle ‘true’ or ‘false’ for each choice.

A. **True** / **False** The interleaving is serializable

B. **True** / **False** The interleaving is permitted under strict two-phase locking

C. **True** / **False** The interleaving is permitted under rigorous two-phase locking

D. **True** / **False** The ordering is valid using snapshot isolation

*Answer.*

A. **F** - The dependency graph has an arrow from READ B in T1 to WRITE B in T2 and an arrow from READ A in T2 to WRITE A in T1, so there is a cycle and therefore the schedule is not serializable

B. **F** - T1 has an S lock on B when T2 is trying to acquire an X lock on B and can’t release its S lock because it has not acquired all its locks yet (still needs to acquire an X lock on A). Therefore T2 can’t acquire the X lock on B

C. **F** - Same as for B, except that T1 would need to have committed in order to release its S lock on B

D. **T** - T1 and T2 have disjoint write sets, so snapshot isolation permits it
VII  ARIES/logging

Consider three transactions that run concurrently, using rigorous two-phase locking:

\[
\begin{array}{ccc}
T1 & T2 & T3 \\
\vdots & \vdots & \vdots \\
RA & RA & RB \\
WA & WA & WB \\
\end{array}
\]

(This is not an interleaving of the operations, just a list of what each transaction does.)

Suppose the system crashes, and T1 has not yet committed. You are given the following partial log at the time of recovery.

<table>
<thead>
<tr>
<th>LSN</th>
<th>Type</th>
<th>TID</th>
<th>PrevLSN</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>BEGIN</td>
<td>T2</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>2</td>
<td>BEGIN</td>
<td>T3</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>3</td>
<td>BEGIN</td>
<td>T1</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>4</td>
<td>UPDATE</td>
<td>T2</td>
<td>-</td>
<td>A</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>COMMIT</td>
<td>T2</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>COMMIT</td>
<td>T3</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

11. [8 points]: Assuming there are no checkpoints or other transactions running and no transactions abort, please fill in all empty cells of the recovery log table above (note that rows with LSN 4-8 all have at least one cell that needs to be filled in).

   \[
   \begin{array}{cccc}
   LSN & Type  & TID & PrevLSN \\
   1   & BEGIN & T2 & - \\
   2   & BEGIN & T3 & - \\
   3   & BEGIN & T1 & - \\
   4   & UPDATE & T2 & 1 \\
   5   & UPDATE & T3 & 2 \\
   6   & COMMIT & T2 & 4 \\
   7   & UPDATE & T1 & 3 \\
   8   & COMMIT & T3 & 5 \\
   \end{array}
   \]

   \textit{Answer.}

Name:
12. [6 points]: Fill in the table below with any log messages that would be added to the recovery log during the UNDO phase of ARIES recovery. Note that the table below starts with LSN 9, immediately following the recovery log above. Additionally, you may not need to use all the rows in the table below.

<table>
<thead>
<tr>
<th>LSN</th>
<th>Type</th>
<th>TID</th>
<th>PrevLSN</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>CLR</td>
<td>1</td>
<td>7</td>
<td>A, 3</td>
</tr>
<tr>
<td>10</td>
<td>EOT</td>
<td>1</td>
<td>9</td>
<td>-</td>
</tr>
</tbody>
</table>

We accepted answers with or without the EOT.
VIII Write-Behind Logging

In this problem, we consider a database recovery system that uses a FORCE/STEAL logging protocol called “Write Behind Logging” (WBL), based on a recent VLDB paper. WBL works as follows:

- The system commits records in rounds, called “epochs”. At each epoch boundary a group of transactions are committed (using group commit). Epochs are numbered with IDs that increment sequentially after the commit is processed.

- As in C-Store and BigTable, the storage system is no overwrite. This means that updates are handled as an insert followed by a delete to the database state on disk. Deletes are processed by associating a pair of epoch IDs with each record, valid and invalid. valid indicates the epoch when the record was committed, and invalid indicates the epoch when a record was deleted (invalid is NULL if the record has not been deleted.) In this way, a record t that has valid = e1 and invalid = e2 is visible in epoch e when e1 ≤ e < e2. Records that are deleted (either via an UPDATE or DELETE) are retained in the storage system (you can assume they are never garbage collected for purposes of this problem).

- Because the system is “FORCE”, modifications to tables for a transaction t are written to disk before the commit record for t is flushed to disk.

13. [10 points]: This system has the surprising property that no log records except COMMIT records need to written, and no UNDO or REDO information is recorded in the log. Describe, in 1–2 sentences, how this can be the case, and how recovery (specifically the UNDO of uncommitted transactions and replay of COMMITTED transactions) would work.

   (Write your answer in the space below.)

Answer. FORCE implies that no REDO (either during runtime or at recovery time) is needed, since all committed transactions have their records on disk. The use of no overwrite storage implies that no UNDO logging is necessary, since the before and after images are on disk, but during recovery time something needs to be done to ensure that records from the last uncommitted epoch are undo. We accepted a variety of answers about how the details of UNDO logging worked, but the ideal answer stated that any record with valid = latest, where latest is the last epoch before the crash, would need to be marked invalid by setting its invalid = latest − 1, and that any record with invalid = latest would need to have its invalid set to NULL.

End of Quiz II

Name: