Department of Electrical Engineering and Computer Science

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

6.814/6.830 Database Systems: Fall 2016 Quiz II

There are 14 questions and **??** 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

1-4 (xx/32)	**5-8(xx/26)**	**9-12 (xx/24)**	**13-14 (xx/18)**	**Total (xx/100)**

Name:

I Distributed Query Processing

Consider a distributed database with a fact table t1 of size 600 MB, and a dimension table t2 of size 50 MB. Users frequently join the two tables on t1.fkey = t2.pkey, and then aggregate the results to produce a small output.

Consider two partitionings:

A : Hash partition t1 on t1.fkey and t2 on t2.pkey

B : Round-robin partition t1, and replicate t2 on every node

1. [8 points]: You try both partitionings on a cluster of 3 nodes where each node has 100 MB of RAM, and you find partitioning B outperforms partitioning A. This is surprising because both configurations need to perform no network I/O for the join, and partitioning A reads strictly less of t2 on every node, since t2 is partitioned instead of replicated. Provide one explanation for how this could occur.
(Write your answer in the space below.)

Answer: Distribution of t1.fkey is skewed.

II Two Phase Commit

Ben Bitdiddle is running a distributed transaction processing system with the two-phase commit (without presumed commit). Ben observes that both log writes and one-way network latencies in his system are about 10 ms, and he finds that the throughput of his system is very low, only about 20 transactions per second. He also notes that most of the transactions in his system do no conflict.

Remembering that group commit is a good way to improve transaction throughput, Ben decides to try adding group commit to the coordinator's write of the commit record. In his system, before writing the commit record, the coordinator waits for X milliseconds for other transactions to be ready to commit, and then writes them to the log as a batch.

2. [6 points]: Do you think this optimization will substantially (by a factor of 2 or more) improve throughput of Ben's system? Why or why not?
(Write your answer in the space below.)

Answer: No, forced log writes on suboords and network latencies will dominate any improvement from logging.

III Locking and OCC

Consider the following transactions, with initial values of X=3 and Y=5. For writes we also indicate the values that are written (i.e., `WY ;Y=tx*2` means that a write to Y is done that sets its value to `tx*2`.)

```
T1              T2              T3
----------      ----------      --------
tx = RX         ty = RY         WY ;Y=0
WY ;Y=tx*2      WX ;X=ty+1
```

3. [8 points]: Suppose you know that the first operation that runs is RY in T2. Assuming there are no deadlocks, list all possible values for X and Y that could result from an execution of these transactions with strict two-phase locking?

(Write your answer in the space below.)

Answer: (1) X = 6, Y = 0. (2) X = 6, Y = 12.

4. [10 points]: Again, assuming the first operation that runs is RY in T2, list all possible values that could arise for X and Y from an execution of these transactions using *snapshot isolation*? Assume snapshot isolation behaves identically to serially-validated OCC, except that read sets are not tracked (so there is no need to check that the read set of a validating transaction intersects with the write set of any concurrent transaction).

(Write your answer in the space below.)

Answer: (1) X = 6, Y = 0. (2) X = 6, Y = 12. (3) X = 6, Y = 6.

IV ARIES/logging

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

```
T1    T2     T3
--    --     --
RA    RA     RA
WA    WA     RC
      RB     RB
      WB     WC
```

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

Suppose the system crashes, and only T1 has committed. You are given following partial log at the time of recovery.

```
1 T1 BEGIN
2 T2 BEGIN
3 T3 BEGIN
4 T1 UPDATE A
5 CHECKPOINT
6 T1 COMMIT
7 T1 END
8 T2 UPDATE A
9 xxx
```

5. [8 points]: Assuming there are no additional checkpoints or other transactions running, what must the value of xxx be?

(Write your answer in the space below.)

Answer: T2 UPDATE B

6. [8 points]: Assuming that there were no transactions running before LSN 1, if T1's update to A is flushed just after LSN 6, and there were no other flushes, at what log record would the REDO phase begin?

(Write your answer in the space below.)

Answer: LSN 4

V Dynamo

In Dynamo, suppose you have three replicas A, B, and C of a data item X, you are running a (N=3,R=2,W=2) system but with sloppy quorums enabled, and with the next successor of A,B,C being D.

Suppose that before each write, the writing client does a (sloppy) quorum read of X, and then performs a write by sending it to a random node in the replica set, supplying one of the most recent versions from the quorum read. When a replica receives a write from a client, it adds or updates its vector clock value and propagates the write to N-1 other nodes (which may not all be in the replica set due to the use of sloppy quorums). The write only succeeds if W nodes acknowledge the write. A node receiving a write to an X that is not causally related to a previous write it received will store both versions of the data (and the write will succeed).

Suppose that replicas process the following sequence of writes, where (A,1) represents the fact that A's vector clock was at 1, and [(A,1),(B,1)] is an example vector clock for X.

1. A creates and successfully writes a new version [(A,1)]
2. B creates and successfully writes a new version [(A,1),(B,1)]
3. C creates and successfully writes a new version [(A,1),(C,1)]

7. [6 points]: Which sets of nodes could have processed write 1?
(Circle True or False for each item below.)

A. True / False A,B, and C

B. True / False just A and B

C. True / False just A and C

D. True / False just A and D

Answer: All of the above.

8. [4 points]: Which sets of nodes could have processed write 2?
(Circle True or False for each item below.)

A. True / False A,B and C

B. True / False just A and B

C. True / False just B and C

D. True / False just B and D

Answer: B, C, D.

9. [4 points]: Which sets of nodes could have processed write 3?
(Circle True or False for each item below.)

A. True / False A,B and C

B. True / False just A and C

C. True / False just B and C

D. True / False just C and D

Answer: All of the above.

10. [4 points]: What causally unrelated versions exist in the system after write 3 completes?
(Write your answer in the space below)

Answer: [(A,1),(B,1)],[(A,1),(C,1)]

VI Spark

You want to analyze the popularity of each website in a huge collection of websites using PageRank. You plan to compute PageRank using Spark on a cluster of 10 machines. All the web pages are stored in HDFS and the following Spark program is run to compute PageRank:

```
val links = spark.textFile(...).map(...).persist() // (URL, links)
var ranks = // RDD of (URL, rank) pairs
for (i <- 1 to ITERATIONS) {
  // Build an RDD of (url, (links, rank)) pairs
  val linksJoinRanks = links.join(ranks);

  // Build an RDD of (targetURL, float) pairs
  // with the contributions sent by each page
  val contribs = linksJoinRanks.flatMap {
    (url, (links, rank)) => links.map(dest => (dest, rank/links.size))
  }
  // Sum contributions by URL
  val sumContribs = contribs.reduceByKey((x,y) => x+y)

  // Get new ranks
  ranks = sumContribs.mapValues(sum => a/N + (1-a)*sum)

}
  // END OF PROGRAM
```

Suppose after the first iteration of the for loop, one machine crashes. Spark will have to recover the failed partitions from the crashed machine.

11. [8 points]: Considering `linksJoinRanks`, `contribs`, `sumContribs` and `ranks`, which RDDs require more work (computation and network communication) to recover? Why?

(Write your answer in the space below.)

Answer: `linksJoinRanks` and `sumContribs` are wide transformation.

Name:

After writing several applications with Spark, you discover that it's possible to append values from an RDD to a file in HDFS. Suppose your replace the line `//END OF PROGRAM` in the previous code with

```
val hdfsFile = // Open a file in append mode on HDFS
ranks.map(x => hdfsFile.append(x))
```

12. [8 points]: You run this program several times (with ITERATIONS set to the same value each time), and you find that you sometimes get different numbers of (URL, rank) pairs in the output HDFS file. Assuming HDFS properly serializes concurrent appends to a file, describe in one sentence how this could happen.

(Write your answer in the space below.)

Answer: If ranks fails, it will re-execute and thus also re-execute any HDFS related operations again

VII Locking

Last year, Ben Bitdiddle built “LocksDB”, a database with a new a custom lock-based concurrency control mechanism. This mechanism extends the read/write locking you saw in class to include increment operations. The core idea of LocksDB is that while reads and writes are able to express increment (or decrement) operations, one can improve concurrency by treating increments differently than arbitrary writes. *The key observation is that the state of the database is the same regardless of the order in which two concurrent increments run.* To get an idea of how transactions with increments work, we provide an example below.

In this listing, `RX` stands for a Read of variable `X`, `WX` stands for a Write of some value to variable `X`, and `IX` means to increment `X` by some value. As in the model of writes presented in class, values used in increments may depend on values previously read. Assume numbers do not overflow and that individual increments happen atomically with respect to other increments/writes.

T1	T2
`IX,5`	
`COMMIT`	
	`local_x = RX`
	`WX,local_x + 5`
	`COMMIT`

Figure 1: Both T1 and T2 are doing the same operation on the data, but T1 uses the increment operation.

For LocksDB, the definition of conflicting operations needs to be extended to include increment operations. Recall that conflicting operations are those where the state of the system depends on the order in which a pair of operations are executed.

In LocksDB, as in the two-phase locking protocol we studied in class, before a transaction can perform a particular R/W/I operation, it needs to hold the appropriate lock, and locks for conflicting operation types cannot be held simultaneously by two transactions. LocksDB has three types of locks: read-only (R), exclusive (X), and increment (I). R and X locks behave as in a conventional 2PL system. I-locks are new.

Below, we have provided the new lock compatibility table that we will use for this exercise. Read locks are compatible with other read locks, and nothing else, and increment locks are compatible with other increment locks, but with nothing else. Exclusive locks are not compatible with anything else.

	R	X	I
R	Y	N	N
X	N	N	N
I	N	N	Y

R and I locks are not compatible because if T1 reads a value and T2 increments it, and T1 re-reads it, it may not read the same value, violating the repeatable reads property.

13. [10 points]: Now that we have the lock compatibility table, we also need to design a modified version of strict 2 phase locking (S2PL). Our modified version of S2PL works the following way: upon receiving a request to read, write or increment an item, the lock manager must decide what lock to acquire, or whether to upgrade an already held lock.

The following table has been partially filled in for you. The rows indicate the lock the transaction currently has on an item, and the columns indicate the lock the transaction should request to perform the operation in the column on the item. Fill in the rest of the entries. Think about what would happen if a transaction attempted to upgrade a lock and another transaction was currently holding the same lock (for example, if T1 and T2 both have R locks on object i, and T1 attempts to write it, T1 would need to acquire an X lock and would be blocked waiting for T2 to release the R lock.)

	R	W	I
hasNone	acqRlock	acqXlock	
hasRlock	nothing	acqXlock	
hasXlock	nothing	nothing	
hasIlock			

Answer: Hint: R can never be promoted to I.

	R	W	I
none	acqRlock	acqXlock	acqIlock
hasRlock	nothing	acqXlock	acqXlock
hasXlock	nothing	nothing	nothing
hasIlock	acqXlock	acqXlock	nothing

14. [8 points]: Ben wants to improve locking performance by allowing dual-granularity locking: a transaction may opt to lock the entire table in R, X or I mode, rather than locking many individual items, or it may lock individual data items. To do this, Ben decides that he will adapt the multi-level locking scheme from class based on *intention locks*. In this scheme, a transaction that wants to read/write/increment an individual data item will first acquire an R/X/I intention lock on the table, and then acquire the appropriate R/X/I on the data item. Fill in the new entries in the lock compatibility table below. IR, IX, II are the intention locks for R, X, and I respectively.

As an example, we've supplied the value for R/IR locks, because a transaction that wants to read some data item inside the table (hence requiring an IR lock on the table) can run concurrently with a transaction that is reading the entire table, the value here is 'Y'.

(Write 'Y' or 'N' in each blank cell.)

	R	X	I	IR	IX	II
R	Y	N	N	Y		
X	N	N	N			
I	N	N	Y			
IR	Y					
IX						
II						

Answer:

	R	E	I	IR	IX	II
R	Y	N	N	Y	N	N
E	N	N	N	N	N	N
I	N	N	Y	N	N	Y
IR	Y	N	N	Y	Y	Y
IX	N	N	N	Y	Y	Y
II	N	N	Y	Y	Y	Y

End of Quiz II