CS127 Homework 7

Due: December 3rd, 2014 2:59 P.M.

Problem 1 (To Be Graded)

Consider the following query on the *account*($aID, name$) and *deposit*($aID, date, amount$) relations:

```
SELECT a.name, d.date, d.amount
FROM deposit AS d
INNER JOIN account AS a
  ON a.aID = d.aID
WHERE amount >= 400
```

Assume that `account` contains 10,000 accounts, and every account has made 50 deposits on average (the `deposit` table contains 500,000 deposits total). Both relations are not sorted in any particular order, and there are no indexes on the relations. The in-memory buffer can hold up to 12 blocks and there is 100 tuples on average in every block (for both tables). Deposits range from $100 (inclusive) to $500 (exclusive), and you may assume an even distribution.

1. Using a **Block Nested-Loop Join**, compute the number of block accesses that will be required to perform the operation.
2. Compute the block accesses again using a **Merge-Join** instead. Both the *deposit* and *account* relations remain unordered.
3. Now let's add some indexes to these tables. First let's add a primary index on *deposit.amount*. The *account* relation remains unordered. Again, using a **Block Nested-Loop Join**, compute the number of block accesses that will be required to perform the operation. Assume that the $WHERE$ clause is evaluated before the $JOIN$, with an index fan out of 100.
4. Assume that the primary index is now changed to *deposit.aID*, rather than *deposit.amount*. The *account* relation remains unordered. Using a **Indexed Nested-Loop Join**, compute the number of block accesses that will be required to perform the operation. Suppose the index fan out is 100.
5. A primary index is added on *account.aID*. Now that both relations are sorted, recompute the number of block accesses using a **Merge-Join**.

Problem 2 (To Be Graded)

Consider the following transactions. Operations are to be executed in order from top to bottom.

Schedule A					
T1	T2	T3	T4	T5	T6
-	-	-	read(G)	-	-
write(H)	-	-	-	-	-
read(G)	-	-	-	-	-
-	-	-	write(H)	-	-
-	-	-	read(C)	-	-
-	-	-	-	-	read(G)
-	-	-	-	read(H)	-
-	-	-	-	write(B)	-
-	-	-	-	-	read(A)
-	write(C)	-	-	-	-
-	-	write(D)	-	-	-
-	-	read(B)	-	-	-
write(E)	-	-	-	-	-
-	-	-	-	-	read(D)
-	-	-	read(E)	-	-
-	-	read(H)	-	-	-
-	-	write(C)	-	-	-
-	-	-	-	-	read(B)

Is this schedule conflict serializable? Prove your answer by building a precedence graph. If serializable, provide a possible serialized order of transactions.

NOTE: Your graph must be large enough (and neat enough) to be read in order to receive credit for this problem. Please provide labels indicating the dependency(s) that each arrow represents.

Problem 3 (To Be Graded)

Assume the following transactions: $T1, T2$ and $T3$:

Schedule			
time	T1	T2	T3
1	-	-	read(A)
2	read(A)	-	-
3	-	-	write(C)
4	-	read(B)	-
5	-	read(C)	-
6	write(B)	-	-

Answer the following questions for the schedule:

1. Is the schedule serializable? Justify your answer by drawing the precedence graph.
2. Assume that all three transactions begin and end at the same time. Could the schedule be produced by the two-phase locking protocol? Insert lock and unlock operations into the schedule to justify your answer.
3. Making the same assumptions, could the schedule be produced by the strict two-phase locking protocol? Insert lock and unlock operations into the schedule to justify your answer.
4. Now assume that the transactions begin at the following timestamps: $T1$: 2, $T2$: 4, $T3$: 1
   All transactions will commit at timestamp 7. Will any transactions in this schedule abort if we are using the time-stamp protocol? Which one(s)?

Problem 4 (To Be Graded)

Consider a database with the following initial values, and the attached command log:
$A = 50, B = 48, C = 0, D = 47$

LOG:
$< T_0, \textbf{start} >$
$< T_0, A, 50, 75 >$
$< T_1, \textbf{start} >$
$< T_1, B, 48, 92 >$
$< T_2, \textbf{start} >$
$< T_2, C, 0, 33 >$
$< T_1, B, 92, 108 >$
$< \textbf{checkpoint} : T_0, T_1, T_2 >$
$< T_3, \textbf{start} >$
$< T_0, A, 75, 100 >$
$< T_2, \textbf{commit} >$
$< T_3, D, 47, 52 >$
$< T_3, \textbf{commit} >$

Assume that the system crashes before the remaining transactions can commit. Using recovery for concurrent transactions, give the following:

1. List any transactions than will need to be undone or redone in the recovery process.
2. List, in order, the set of logged operations to be performed to undo or redo the transactions. (*i.e. "Set A to 7", "Set B to 39", etc.*)
3. Give the final values for A, B, C, and D.