CSCI 6333/6315 Database
Final Exam, Spring 2020

Name: ______________________

ID: ______________________

Note: *Please type your answers. If you need to draw a figure, you may draw on a paper, take a picture, and then cut and paste the figure to your file.*

Problem 1 [20 points]. Consider the schema $R = (A, B, C, D, E, F)$ and the following set $\boldsymbol{F}$ of functional dependencies holds on R:

$$A \longrightarrow BCD$$
$$BC \longrightarrow DE$$
$$B \longrightarrow D$$
$$D \longrightarrow A$$

a) Compute A^+.
   Answer:

$$\begin{aligned} A &\Longrightarrow A,\ \ A \longrightarrow A \\ &\Longrightarrow ABCD,\ \ A \longrightarrow BCD \\ &\Longrightarrow ABCDE,\ \ BC \longrightarrow DE \end{aligned}$$

   Hence, $A^+ = ABCDE$.

b) Prove that the decomposition $R_1 = (A, B, C, F)$ and $R_2 = (B, D, E, F)$ of R is a lossless-join decomposition.

   Answer:

$$R_1 \cap R_2 = BF$$

$$\begin{aligned} BF &\Longrightarrow BF,\ \ BF \longrightarrow BF \\ &\Longrightarrow BDF,\ \ B \longrightarrow D \\ &\Longrightarrow ABDF,\ \ D \longrightarrow A \\ &\Longrightarrow ABCDF,\ \ A \longrightarrow BCD \\ &\Longrightarrow ABCDEF,\ \ BC \Longrightarrow DE \end{aligned}$$

   Hence, $(BF)^+ = ABCDEF$, which implies $BF \longrightarrow R_1$ and $BF \longrightarrow R_2$, therefore the decomposition is lossless.

Problem 2. [20 points] Let the relations $r_1(A, B, C)$ and $r_2(C, D, E)$ have the following properties: r_1 has 30,000 tuples, r_2 has 45,000 tuples, 25 tuples of r_1 fit on one block, and 30 tuples of r_2 fit on one block. Estimate the number of block accesses required, using each of the following join strategies for $r_1 \bowtie r_2$ to determine the efficient loop structure, that is, which relation shall be used for the outer loop (and the other is for the inner loop).

a) Nested-loop join
b) Block nested-loop join

Solution:

$$n_{r1} = 30000, \qquad n_{r2} = 45000, \qquad b_{r1} = \frac{n_{r1}}{25} = 1200, \qquad b_{r2} = \frac{n_{r2}}{30} = 1500$$

a) When r1 is for the outer and r2 is for the inner loop, the estimated cost is $n_{r1} * b_{r2} + b_{r1} = 30000 * 1500 + 1200 = 45{,}001{,}200$ block transfers, plus $n_{r1} + b_{r1} = 30{,}000 + 1{,}200 = 31{,}200$ seeks.

   When r2 is for the outer loop and r1 is for the inner loop, the estimated cost is $n_{r2} * b_{r1} + b_{r2} = 45{,}000 * 1{,}200 + 1{,}500 = 54{,}001{,}500$ block transfers, plus $n_{r2} + b_{r2} = 45{,}000 + 1{,}500 = 46{,}500$ seeks.

   It is more efficient to have r1 for the outer loop and for r2 for the inner loop.

b) When r1 is for the outer and r2 is for the inner loop, the estimated cost is $b_{r1} * b_{r2} + b_{r1} = 1{,}200 * 1{,}500 + 1{,}200 = 1{,}801{,}200$ block transfers, plus $2 * b_{r1} = 2 * 1{,}200 = 2{,}400$ seeks.

   When r2 is for the outer loop and r1 is for the inner loop, the estimated cost is $b_{r2} * b_{r1} + b_{r2} = 1{,}500 * 1{,}200 + 1{,}500 = 1{,}801{,}500$ block transfers, plus $2 * b_{r2} = 2 * 1{,}500 = 3{,}000$ seeks.

   It is more efficient to have r1 for the outer loop and for r2 for the inner loop.

Problem 3. [10 points] Give two concrete examples to show that there are schedules that are possible under the two-phase locking protocol, but are not possible under the timestamp protocol, and ***vice versa***.

Answer:

a) A 2PL protocol but not timestamp

T3	T4
Lock(A)	
W(A)	
	Lock(B)
	W(B)
	Unlock(B)

Lock(B)
R(B)
Unlock(B)
Unlock(A)

b) A timestamp protocol but not 2PL

T5	T6	
W(A)		
W(B)		
		← for 2PL, T5 must unlock B so that T6 lock B and then write B
	W(B)	
		← for 2PL, T5 must unlock A so that T6 can lock A and then read A
	R(A)	
		← for 2PL, T6 must unlock A so that T5 can lock A and then read A This contradicts to 2PL for T5
R(A)		

Problem 4. [10 points] When the system recovers from a crash, it constructs an undo-list and a redo-list. Explain why log records for transactions on the undo list must be processed in reverse order, while those log records for transactions on the redo-list are processed in a forward direction.

Answer:

Consider the following log records:

<T1, x, 10, 20>
<T1, x, 20, 30>
<T1, x, 30, 40>

To undo, we'd like to restore the oldest value 10 to x. If we do so in forward order, the value of x will be 30. If we do so in backward order, x will be correctly set to 10.

To redo, if we follow the backward order, we'll set x to 20, which shall be 40. If we follow the forward order, we'll correctly set x to 40.

Problem 5. [10 points] Let R be a relation and F a set of functional dependencies of R. |R|= n and |F= m.

a) What is the time complexity to check whether R is in BCNF?
b) What is the time complexity to check whether R is in 3NF?

Answer:

a) Following the definition of BCNF, we need to check, for any nontrivial functional dependency $\alpha \to \beta$ in F, whether $\alpha \to R$, i.e., whether α is superkey for R. To do so, we need to compute α^+ and to check whether it contains R. We need in the worst case to compute one attribute closure for every functional dependency in F. Since the time complexity of computing an attribute closure is $O(mn^2)$, so the total time needed is $O(m^2n^2)$.

b) Unlike the problem of testing whether R is BCNF, the problem of testing R is in 3NF is hard. For any nontrivial functional dependency $\alpha \to \beta$ where α is not a superkey for R, the third condition of the 3NF definition asks for each attribute *A* in $\beta - \alpha$ whether A is contained in a candidate key for *R*. This implies that to facilitate the testing for the third conditions, we need to find candidate keys for R, which is NP-hard. Hence, the time complexity of testing whether R is in 3NF is NP-hard.

Problem 6. [10 points] A disk block has 1024 bytes, and both a pointer and a search key are of 4 bytes each. Now, consider that we build a B^+-tree index with a disk block to store every tree node. Assume that the index tree has a height of 4. Estimate the number of unique search key values that can be represented by the index tree.

Answer:

Following the given information, a bock can store 1024/4 = 256 pointer and/or search key values. Thus, with the size of a disk block, each internal node can store 128 pointers and 127 search key values, leaving four bytes wasted.

Each internal node has at least $\lceil 128/2 \rceil = 64$ children and at most 128 children. With height 4, the B^+-tree index has at least 64^4 leaf nodes and at most 128^4 leaf nodes.

At leaf a node, the 4 bytes wasted in an internal node can be used to point the next leaf node. By this way, each leaf node has at least $\lceil 128/2 \rceil = 64$ search key values and at most 128^4 search key values.

Therefore, with height 4, the B^+-tree index will have at least $64^5 =$ 1,073,741,824 search key values and at most $128^5 = 34{,}359{,}738{,}368$ search key values.

Problem 7 [20 points] Show that the wound-wait strategy will prevent

a) Deadlock, and
b) Starvation.

Proof.

Let $T_1, T_2, \dots, T_n$ be a list of transactions. Suppose that they follow the wound-wait strategy with timestamp $t(T_i) = i$. We have the following two claims.

Claim 1: There is no deadlock.

Proof of Claim 1. We construct a wait-for graph G for $T_1, T_2, \dots, T_n$. Assume by contradiction that there is a deadlock. That is, G has a cycle $T_{i_1} \to T_{i_2} \to \cdots \to T_{i_j} \to T_{i_1}$. Because all transactions follow the wound-wait strategy, and this strategy only allows a younger transaction to wait for an older transaction, the cycle implies

$$t(T_{i_1}) > t(T_{i_2}) > \cdots > t(T_{i_j}) > t(T_{i_1}).$$

Thus, we have a contradiction. Hence, there is no deadlock.

Claim 2: There is no starvation.

Proof of Claim 2. Let $f(T_i)$ denote the time at which transaction T_i is completely executed. Let us consider T_1 first. Note that T_1 is the oldest transaction. When T_1 became active, there are two possibilities: (a) There were no other active transactions; and (b) There were some active transactions. For (a), since T_1 is the oldest transaction, it never be wounded by any other transaction, so it will continue execution till completion. Thus, it will not starve and will finish at time $f(T_1)$. For (b), since T_1 is the oldest, by the wound-die strategy it will wound all active transactions to force them to abort so as to gain its execution. Thus, this enters case (a). Hence, T_1will not starve and finish at time $f(T_1)$.

For transaction T_2, it may be completed before $f(T_1)$. If not, then after $f(T_1)$, T_2 becomes the oldest transaction and, by the similar analysis for T_1, T_2 it will not starve and complete its execution at time $f(T_2)$. We can carry out the similar analysis for $T_3, \dots, T_n$, and none of them will starve.