CS 245
Final Exam – Winter 2002

This exam is open book and notes. You have 120 minutes to complete it.

Print your name:____

The Honor Code is an undertaking of the students, individually and collectively:

1. that they will not give or receive aid in examinations; that they will not give or receive unpermitted aid in class work, in the preparation of reports, or in any other work that is to be used by the instructor as the basis of grading;
2. that they will do their share and take an active part in seeing to it that others as well as themselves uphold the spirit and letter of the Honor Code.

The faculty on its part manifests its confidence in the honor of its students by refraining from proctoring examinations and from taking unusual and unreasonable precautions to prevent the forms of dishonesty mentioned above. The faculty will also avoid, as far as practicable, academic procedures that create temptations to violate the Honor Code.

While the faculty alone has the right and obligation to set academic requirements, the students and faculty will work together to establish optimal conditions for honorable academic work.

I acknowledge and accept the Honor Code.

Signed:____

Problem	Points	Maximum
1		10
2		10
3		20
4		10
5		10
6		20
7		20
8		20
Total		120

Problem 1 (10 points)

(a) Show the results of entering the keys 1, 2, ..., 10 (in that order) to an initially empty B+-tree with order $n = 3$. In case of overflow, split the block (do not re-distribute keys to neighbors).

RESULTING TREE:

Root(7)

Level 2 nodes: (3,5) and (9)

Level 3 nodes: (1,2) (3,4) (5,6) (7,8) (9,10)

(b) What is the utilization of the tree? To compute utilization, divide the number of existing keys in the tree (across all levels, including leaves), by the number of possible keys that could fit in the same tree (across all levels, including leaves).

UTILIZATION:14/24

(c) Now demonstrate a *different* insertion order that leads to a tree of different depth than the one in part (a).

INSERTION ORDER:1,2,4,5,3,7,8,6,9,10

RESULTING TREE:

Root: (4 7 9) Children: (1 2 3) (4 5 6) (7 8) (9 10)

(d) What is the utilization of the new tree?

UTILIZATION:13/15

Problem 2 (10 points)

Say you have a relation with 100,000 records. You want to hash the relation into a hash table with 1000 buckets. A disk block can store at most 100 records (along with an optional pointer to an overflow block). Assume that a disk block cannot store records from two different buckets.

(a) What is the maximum number of disk blocks you would need for the relation?

ANSWER: 1990 blocks. There can be at most one non-full block per bucket. Thus, in the worst case, we would have 1000 blocks with 1 record each, and the remaining 99,000 records would fit on 990 blocks. Therefore, the total number of blocks needed is 1990.

Common Errors: Assuming that empty buckets need a disk block. The question asks for the number of blocks you would need for the relation, not the number of blocks possibly allocated to the hash table when using a less-than-optimal block-allocation policy.

(b) What is the minimum number of disk blocks you would need?

ANSWER: 1000 blocks. We have 100,000 records and each block can store 100 records. So, we need 1000 blocks.

(c) What is the answer to (a) if the relation had 100,099 records?

ANSWER: 1990 blocks. Consider the worst-case scenario of (a). If we add 99 more records to such a hash table, we wouldn't need to allocate any new blocks since every bucket can store up to 99 more records. Another way to look at it is to observe that we can still have at most 1000 non-full blocks. If we assume there are a total of $x (\geq 1000)$ records in these non-full blocks, $(100,099 - x)$ records need to fit exactly in a whole number of blocks, which means that $(100,099 - x)$ must be divisible by 100. The smallest x for which this happens is 1099.

Next, suppose that you want to store the relation with 100,000 records in a B+ tree. However, unlike a standard B+ tree, here the leaf nodes store the actual records, as opposed to just pointers to the records. Thus, a leaf node can store a maximum of 100 records. That is, for leafs the order is 100. For non-leaf nodes, the order is 1000, i.e., a maximum of 1000 keys can be stored.

(d) What is the maximum number of disk blocks you would need to store the relation in this B+ tree?

ANSWER: 2004 blocks.

In the worst case, the leaf blocks are all half-full and we thus need $100,000/50 = 2000$ leaf blocks. Every node above the leaf needs to have at least 501 leaf pointers. Since there are a total of 2000 leaf blocks, we could have at most 3 interior nodes pointing to the leaves. We would then need a root node on top of these 3 interior nodes, for a total of 2004 blocks.

Common Errors: Believing that there could be 4 interior nodes pointing to the leaf blocks. If there were 4 interior nodes, at least one of them must have 500 or fewer pointers, which would be a violation of the B-tree condition.

(e) What is the minimum number of disk blocks you would need?

ANSWER: 1001. When all leaf blocks are full, we would have 1000 leaf blocks. We need one root node to point to these 1000 leaf blocks, for a total of 1001 blocks.

Problem 3 (20 points)

Consider the following database stored on disk:

Element	Value
A	13
B	40
C	35
D	4
E	18

For each of the following logs state (i) whether that log could be an undo log for actions that resulted in the above database, and (ii) whether that log could be a redo log for actions that resulted in the above database. For each of (i) and (ii), if not, explain why not.

(a)
```
<START T1>
<T1,C,35>
<T1,D,450>
<START T2>
<T2,C,18>
<T2,B,40>
<COMMIT T1>
<START CKPT (T2)>
<END CKPT>
<T2,D,18>
<START T3>
<T3,C,35>
<T3, E,18>
<T2,A,13>
<COMMIT T3>
<COMMIT T2>
```

COULD BE UNDO LOG?: NO

In an undo log, all transaction that are active at the beginning of a checkpoint need to commit by the end of the checkpoint. Since the checkpoint here ends before T_2 commits, this log cannot be an undo log.

COULD BE REDO LOG?: NO

If this log were a redo log, the value of D should either be 450 (since T_1 commits before the start of the checkpoint) or 18 (which is written by T_2). Since the final value of D in the database is 4, the log cannot be a valid redo log.

(b)

```
<START T1>
<T1,D,4>
<START T2>
<T2,E,6>
<T1,A,5>
<START CKPT (T1,T2)>
<T1,E,18>
<START T3>
<T3,C,35>
<T3,A,13>
<COMMIT T2>
<T3,B,40>
<COMMIT T3>
<END CKPT>
<T1,A,11>
<COMMIT T1>
```

COULD BE UNDO LOG?: NO

This log cannot be an undo log for the same reason as in part (a).

COULD BE REDO LOG?: YES

Problem 4 (10 points)

Imagine that you have a data warehouse with the following relations:

- Customers(Name,Age,Gender,CustID): 100,000 disk blocks
- Purchases(CustID,Product,Date,Location,Amount): 2,000,000 disk blocks
- SalesCalls(CustID,Salesperson,Date,Result): 300,000 disk blocks

You have observed the following query mix over these relations:

- 10% queries selecting on Customers.CustID
- 30% queries selecting on Customers.Name
- 35% queries selecting on Purchases.Product
- 10% queries selecting on SalesCalls.SalesPerson
- 15% queries selecting on SalesCalls.Date

You want to create indexes over these relations to speed up queries over these relations. You decide that you have enough resources (disk space, etc.) to build two indexes. You may assume that the index allows you to retrieve the answer to the query with significantly less cost than doing a table scan.

Which attributes should you build indexes over? Please explain briefly.

ANSWER: Purchases.Product and SalesCalls.Date

The savings obtained by building an index on an attribute is proportional to the number of blocks in the corresponding relation multiplied by the number of queries using that index. Purchases.Product is clearly the best choice, since Purchases is the largest relation and Purchases.Product is the most queried attribute. For the second index, we have to choose between Customers.Name and SalesCalls.Date. Of these two, SalesCalls.Date is more useful since $0.15 * 300000 > 0.1 * 100000$.

Problem 5 (10 points)

Consider a database that uses a validation mechanism for concurrency control. There are only 5 kinds of transactions in the system. The read, write actions of the 5 kinds of transactions are given below.

(1) r(A) r(B) w(A) w(C)

(2) r(B) w(D) w(E)

(3) r(B) r(D) w(F)

(4) r(B) r(E)

(5) r(B) r(C) w(F)

Note that (1) – (5) are transaction types and not transactions. In particular we can have two different transactions of the same type. Two transactions are said to execute concurrently if at some point of time, both the transactions have started, but neither has finished.

(a) Determine for each pair of transaction types (m, n) (including the case $m = n$), if the transaction of type n can be aborted due to a transaction of type m when executing concurrently (in the absence of any other transaction). Please indicate your answer in the table below by placing a YES in column n and row m if a transaction of type n can be aborted due to a transaction of type m, and NO otherwise.

	Transaction of this type (n)				
Can be aborted by by type below (m)	(1)	(2)	(3)	(4)	(5)
(1)	YES	NO	NO	NO	YES
(2)	NO	YES	YES	YES	NO
(3)	NO	NO	YES	NO	YES
(4)	NO	NO	NO	NO	NO
(5)	NO	NO	YES	NO	YES

We place a "YES" in the table at row m and column n if the write set of transaction type m intersects with the read or write sets of transaction type n.

(b) We now want to determine if two transactions of types n and m can be run without any concurrency control. The answer will be true if under every possible schedule, neither transaction can cause the other to abort (in the absence of any transactions of types different from n and m).

Write an expression $E(n, m)$ that tells us when transactions can be run without concurrency control. In particular, expression $E(n, m)$ should evaluate to true if transactions of types n and m can be run without concurrency control. Write your answer using the table of part (a), where TABLE(n, m) is true if entry (n, m) in the table is YES (true). (For example, your answer can be an expression like "TABLE(n, m) AND NOT TABLE$(n, m + 1)$," but this is of course *not* the right answer.)

ANSWER: NOT (TABLE(m,n) OR TABLE(n,m))

Problem 6 (20 points)

Consider a relational database with tuples $A, ...L$, stored in disk blocks $1, ...4$ as shown:

- Block 1: A, B, C
- Block 2: D, E, F
- Block 3: G, H, I
- Block 4: J, K, L

Assume we only have one kind of lock: exclusive locks. Define a "convoy" as a point in time in which one transaction T holds a lock on an object O, and at least two other transactions are waiting for the lock on object O. Define a "deadlock" as a point in time in which there is a sequence of transactions $T_1, ...T_n$, such that for all i such that $i < n$, T_i waits for T_{i+1}, and also T_n waits for T_1.

(a) Assume that transactions can acquire locks on individual tuples ("tuple level locking"), and consider the following three transactions:

- T_1 : $L(E)R(E)L(H)R(H)W(E)UL(E)UL(H)$
- T_2 : $L(A)R(A)L(E)R(E)W(A)UL(A)UL(E)$
- T_3 : $L(G)R(G)L(D)R(D)W(D)UL(D)UL(G)$

Note: L = lock, R = read, W = write, UL = unlock.

Is there some schedule where a convoy occurs? If so, draw the waits-for graph that shows the convoy. If not, explain why not.

ANSWER: No convoy can occur, since no one tuple is locked by more than two transactions.

(b) For the same scenario as part (a), is there some schedule where a deadlock occurs? If so, draw the waits-for graph that shows the deadlock. If not, explain why not. Label the arcs in a waits-for graph with the object (tuple) that is being waited for.

ANSWER: No deadlock can occur. Only one tuple (E) is locked by two different transactions, and there must be at least two different tuples locked by multiple transactions for there to be a deadlock.

(c) Now assume that transactions can acquire locks only on whole blocks. For example, to lock tuple A, a transaction must actually acquire a lock on block 1. For the same set of transactions as part (a), is there a *new* convoy (i.e., a new set of transactions) that could occur now that could not occur under tuple-level locking? If so, draw the waits-for graph that illustrates the convoy.

ANSWER:

$$T_2 \xrightarrow{2} T_1 \xleftarrow{2} T_3$$

(d) With block-level locks and the transactions of part (a), is there a *new* deadlock (i.e., a new set of transactions) that could occur now that could not occur under tuple- level locking? If so, draw the waits-for graph that illustrates the deadlock. Again, label the arcs in a waits-for graph with the object (block) that is being waited for.

ANSWER:

$$T_1 \underset{3}{\overset{2}{\leftrightarrows}} T_3$$

(e) Next we consider a different set of transactions and the same questions posed in parts (a) through (d). Consider the transactions

- T1: L(E)R(E)L(H)R(H)W(E)UL(E)UL(H)
- T2: L(J)R(J)L(E)R(E)W(E)UL(E)UL(J)
- T3: L(C)R(C)L(H)R(H)L(J)R(J)W(H)UL(H)UL(J)UL(C)
- T4: L(K)R(K)L(E)R(E)W(E)UL(E)UL(K)

With record-level locking, is there some schedule where a convoy occurs? If so, draw the waits-for graph that shows the convoy. If not, explain why not.

ANSWER:

$$T_2 \xrightarrow{E} T_1 \xleftarrow{E} T_4$$

(f) For the same scenario as part (e), is there some schedule where a deadlock occurs? If so, draw the waits-for graph that shows the deadlock. If not, explain why not. Label the arcs in a waits-for graph with the object (tuple) that is being waited for.

ANSWER:

$$\begin{tikzcd} T_1 \arrow[rr, "H"] & & T_3 \arrow[dl, "J"] \\ & T_2 \arrow[ul, "E"] & \end{tikzcd}$$

(g) Again assume that transactions can acquire locks only on whole blocks. For the same set of transactions as part (e), is there a *new* convoy (i.e., a new set of transactions) that could occur now that could not occur under tuple- level locking? If so, draw the waits-for graph that illustrates the convoy.

ANSWER:

$$T_2 \xrightarrow{4} T_4 \xleftarrow{4} T_3$$

(h) With block-level locks and the transactions of part (e), is there a *new* deadlock (i.e., a new set of transactions) that could occur now that could not occur under tuple-level locking? If so, draw the waits-for graph that illustrates the deadlock. Again, label the arcs in a waits-for graph with the object (block) that is being waited for.

ANSWER:

$$\begin{tikzcd}
T_1 \arrow[rr, "3"] & & T_3 \arrow[dl, "4"] \\
& T_4 \arrow[ul, "2"] &
\end{tikzcd}$$

Problem 7 (20 points)

Consider a multi-dimensional cube based on a fact table $R(A, B, C, M)$, where A, B, and C and the dimensions and M is the measure. The aggregation operation is SUM.

In the fact table R we know that

- $V(R, A) = 10$
- $V(R, B) = 50$
- $V(R, C) = 1000$.

We also know that for every combination of A, B, C values there is exactly one tuple in R. (That is, there are $10 \times 50 \times 1000$ tuples in R.)

To answer queries efficiently, we have materialized the cubes

- $(A, B, *, M)$
- $(A, *, C, M)$
- $(*, *, C, M)$.

(a) Suppose that we want to answer the query $(a_1, *, *, M)$ for some value of $A = a_1$. How many additions do we need to perform if we start from

1. The original fact table R?

   ANSWER: 49,999. Observe that the number of additions needed is one less than the number of elements to be added.

2. The materialized cube $(A, B, *, M)$?

   ANSWER: 49

3. The materialized cube $(A, *, C, M)$?

   ANSWER: 999

Thus, how should we compute the answer to $(a_1, *, *, M)$?

ANSWER: Use $(A, B, *, M)$.

(b) Next, suppose that we want to answer the query $(*, *, *, M)$. Explain how this query should be answered in the most efficient fashion.

ANSWER: Use (A,B,*,M).

(c) Next, suppose that we have a dimension hierarchy for attribute B: B – BD_1 – BD_2 – all. (For example, if B is "city," then BD_1 can be "state" and BD_2 can be "country.") Suppose that we have materialized cube (A, BD_1, C, M). List all other cubes that can be computed from (A, BD_1, C, M) in one step, i.e., by aggregating a single attribute. (These are the cubes that are neighbors of (A, BD_1, C, M) in the lattice of cubes.)

ALL CUBES: $(*, BD_1, C, M)$, (A, BD_2, C, M), $(A, *, C, M)$ and $(A, BD_1, *, M)$.

Problem 8 (20 points)

Consider three transactions

- $T_1: \ r(a)\ r(b)\ w(a)\ w(b)\ w(c)$
- $T_2: \ r(c)\ r(b)\ w(b)$
- $T_3: \ r(c)\ r(a)\ w(c)$

Each part below asks for a schedule of a particular type, for transactions T_1, T_2, T_3 (and no others). Please represent your schedules in a 2-D grid, with time flowing down the vertical axis and a separate column for each data value. Also, please show the validation or commit points in each schedule. *Careful: In some cases, there may be no schedule satisfying the requested properties. If there is no schedule, simply write "NO SCHEDULE EXISTS."*

Since there are slight differences between the definitions in the book and in the class notes, **for this problem use the difinition in the class notes:**

- Schedule S is recoverable if each transaction commits only after all transactions from which it read have committed.
- Schedule S avoids cascading rollback if each transaction may read only those values written by committed transactions.
- Schedule S is strict if each transaction may read and write only items previously written by committed transactions.

(a) A serializable, but not recoverable schedule generated by a locking based scheduler.

a	b	c	commit point
r_1			
	r_1		
w_1			
	w_1		
		w_1	
		r_2	
	r_2		
	w_2		
			C_2
			C_1
		r_3	
r_3			
		w_3	
			C_3

The schedule shown above is not the only possible solution to the problem but is representative of all the solutions. In order for a schedule to be non-recoverable, we need to ensure that there are two transactions T_x and T_y such that T_x reads a value written by T_y and T_x commits before T_y. In the solution shown above we have used $T_x = T_2$ and $T_y = T_1$.

(b) A serializable, but not recoverable schedule generated by a validation scheduler.

NO SCHEDULE EXISTS. All schedules generated by validation schedulers are recoverable. (See part (c) for explanation.)

(c) A serializable and recoverable schedule, that does *not* avoid cascading rollback, generated by a validation scheduler.

NO SCHEDULE EXISTS. All schedules generated by validation schedulers avoid cascading rollback. The algorithm that the scheduler uses for validating transactions aborts transactions that might possibly have read dirty data. Therefore, all validated transactions read only committed data and, consequently, avoid cascading rollback.

(d) A serializable, recoverable, ACR and strict schedule, but *not* serial, generated by a validation scheduler.

a	b	c	validation point
r_1			
	r_1		
			V_1
w_1			
	w_1		
		w_1	
		r_2	
	r_2		
		r_3	
			V_2
r_3			
			V_3
	w_2		
		w_3	

Observe that all schedules generated by validation schedulers are serializable, recoverable, ACR and strict. All the question requires is to produce a non-serial schedule. Such a schedule is easily generated by interleaving T_2 and T_3, ensuring that T_2 validates before T_3. An alternative solution would be to interleave T_3 with T_1 and ensuring that T_3 *finishes* before T_1 validates.

(e) A schedule that *cannot* be generated by a validation-based scheduler, but can be generated by a strict, locking-based scheduler.

a	b	c	commit point
r_1			
		r_2	
	r_2		
	w_2		
			C_2
	r_1		
w_1			
	w_1		
		w_1	
			C_1
		r_3	
r_3			
		r_3	
			C_3

The schedule shown above is strict, but cannot be generated by a validation-based scheduler. Any validation scheduler would abort either T_1 or T_2 since there are read-write conflicts between the two in both directions, i.e., $RS(T_1) \cap WS(T_2) \neq \phi$ and $RS(T_2) \cap WS(T_1) \neq \phi$.