Department of Electrical Engineering and Computer Science

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

6.830 Database Systems: Fall 2008 Quiz I Solutions

There are 17 questions and 13 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!

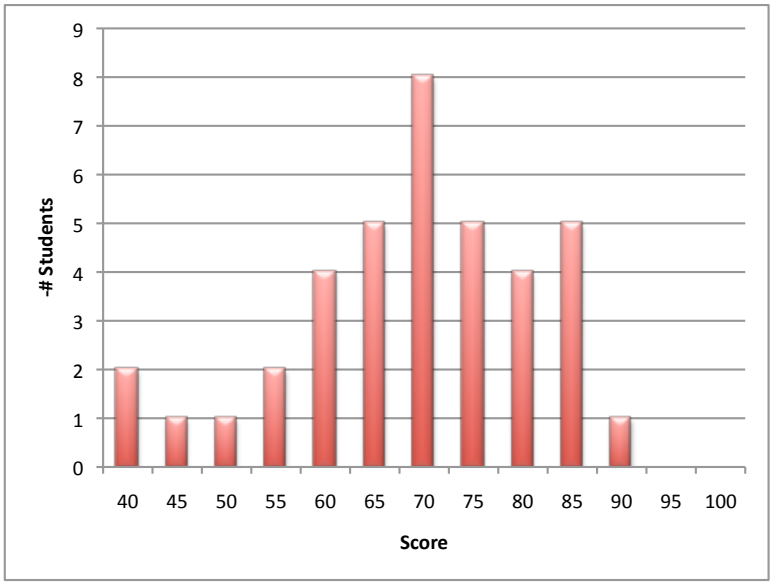
THIS IS AN OPEN BOOK, OPEN NOTES QUIZ.
NO PHONES, NO LAPTOPS, NO PDAS, ETC.

Do not write in the boxes below

1-4 (xx/20)	5-7 (xx/16)	8-10 (xx/20)	11-13 (xx/21)	14-17 (xx/23)	Total (xx/100)

Name: Solutions

μ = 71.2
σ = 12.4
25% = 64.25
50% = 73
75% = 82.5



Score histogram

I Short Answer

1. [6 points]: To reduce the number of plans the query optimizer must consider, the Selinger Optimizer employs a number of heuristics to reduce the search space. List three:
(Fill in the blanks below.)

- Push selections down to the leaves.
- Push projections down.
- Consider only left deep plans.
- Join tables without join predicates (cross product joins) as late as possible.

2. [4 points]: Give one reason why the REDO pass of ARIES *must* use physical logging.
(Write your answer in the space below.)

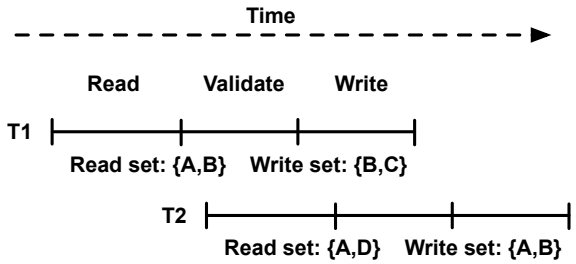
The pages on disk may be in any state between "no updates applied" and "all updates applied." Logical redo cannot be used on an indeterminate state, whereas physically redo logging just overwrites the data on disk with the logged values. In other words, physical logging is idempotent whereas logical logging is not.

Name:

II Optimistic Concurrency Control

For the following transaction schedules, indicate which transactions would commit and which would abort when run using the parallel validation scheme described in the Kung and Robinson paper on Optimistic Concurrency Control. Also give a brief justification for your answer. You may assume that if a transaction aborts, it does not execute its write phase, rolling back instantly at the end of the validation phase.

3. [4 points]:



(Write your answer in the spaces below.)

Transactions that commit: T1,T2 or T1

Transactions that abort: {} or T2

Justification:

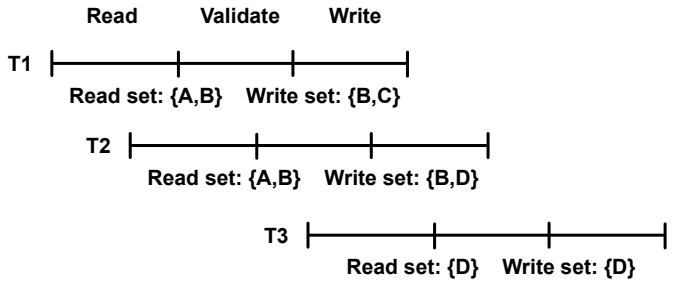
There are two correct solutions:

T1 and T2 both commit. T1 commits because it validates first. T2 will also commit because while its write set intersects T1's write set, T1 finishes its write phase before T2 starts its write phase (condition 2 in section 3.1 of the paper).

T1 commits, T2 aborts. T1 commits because it validates first. In the pseudocode from the paper (section 5), T1 is executing the line marked "write phase" when T2 starts its validation phase. T2 finds T1 in the "finish active" set, and aborts due to a write/write conflict. This is sort of a "false" conflict, since it would be safe for T2 to commit. However, the parallel algorithm assumes that T1 is still writing at the time that T2 detects the conflict, so it must abort.

Name:

4. [6 points]:



(Write your answer in the spaces below.)

Transactions that commit: T1,T3

Transactions that abort:T2

Justification:

T1 and T3 commit, T2 aborts. T1 commits because it validates first. T2 must abort because T1's write set intersects T2's read set, and T2 was started reading before T1 finished writing. T3 then commit because although it has a conflict with T2, when it begins the validation algorithm, T2 has already aborted (it finishes aborting at the end of the validation phase).

Name:

III Schema Design

Consider a relational table:

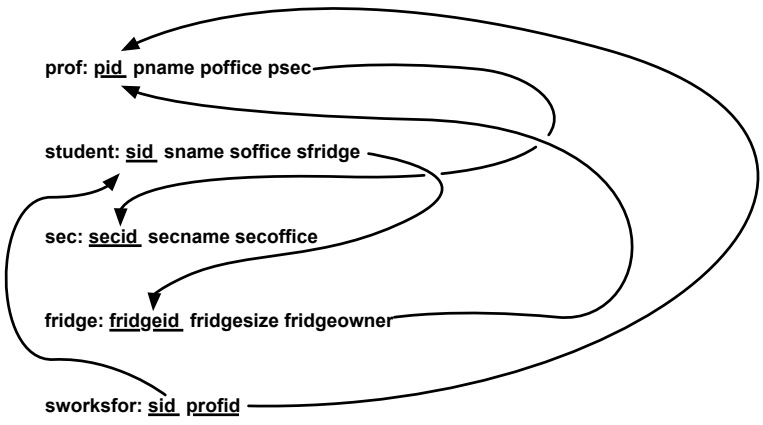
```
Professor(
   professor_name, professor_id,
   professor_office_id, student_id,
   student_name, student_office_id,
   student_designated_refrigerator_id, refrigerator_owner_id,
   refrigerator_id, refrigerator_size, secretary_name,
   secretary_id, secretary_office )
```

Suppose the data has the following properties:

- **A.** Professors and secretaries have individual offices, students share offices.
- **B.** Students can work for multiple professors.
- **C.** Refrigerators are owned by one professor.
- **D.** Professors can own multiple refrigerators.
- **E.** Students can only use one refrigerator.
- **F.** The refrigerator the student uses must be owned by one of the professors they work for.
- **G.** Secretaries can work for multiple professors.
- **H.** Professors only have a single secretary.

5. [10 points]: Put this table into 3rd normal form by writing out the decomposed tables; designate keys in your tables by underlining them. Designate foreign keys by drawing an arrow from a foreign key to the primary key it refers to. Note that some of the properties listed above may not be enforced (i.e., guaranteed to be true) by a 3NF decomposition.

(Write your answer in the space below.)



Name:

6. [3 points]: Which of the eight properties (A–H) of the data are enforced (i.e., guaranteed to be true) by the 3NF decomposition and primary and foreign keys you gave above?

(Write your answer in the space below.)

B,C,D,E,G,H; the schema does not enfore that profs/secretaries have individual offices or that students use the fridge of the professor they work for.

7. [3 points]: What could a database administrator do to make sure the properties not explicitly enforced by your schema are enforced by the database?

(Write your answer in the space below.)

Use triggers or stored procedures to ensure that these constraints are checked when data is inserted or updated.

IV External Aggregation

Suppose you are implementing an aggregation operator for a database system, and you need to support aggregation over very large data sets with more groups than can fit into memory.

Your friend Johnny Join begins writing out how he would implement the algorithm. Unfortunately, he stayed up all night working on Lab 3 and so trails off in an incoherent stream of obscenities and drool before finishing his code. His algorithm begins as follows:

input : Table T, aggregation function $fname$, aggregation field $aggf$, group by field $gbyf$
`/* Assume` T `has` $|T|$ `pages and your system has` $|M|$ `pages of memory */`

`/* Partition phase */`
$n \leftarrow \left\lceil \frac{|T|}{|M|} \right\rceil$;
Allocate $bufs$, n pages of memory for output buffers ;
Allocate $files$, an array of n files for partitions ;
foreach record $r \in T$ **do**
  $h \leftarrow hash(r.gbyf)$; /* $h \in [1 \ldots n]$ */
  Write r into page $bufs[h]$;
  if page $bufs[h]$ is full **then**
    Add $bufs[h]$ to file $files[h]$, and set the contents of $bufs[h]$ to empty ;
  end
end

`/* Aggregate phase */`
foreach $f \in files$ **do**
  `/* Your code goes here. */`
end

Name:

8. [6 points]: Relative to $|T|$, how big must $|M|$ be for the algorithm to function correctly? Justify your answer with a sentence or brief analytical argument.

(Write your answer in the space below.)

Because $bufs$ should fit in memory, $n \leq |M|$.

The number of hash buckets is computed as $n = \left\lceil \frac{|T|}{|M|} \right\rceil$.

Plugging in for n, we get that $\left\lceil \frac{|T|}{|M|} \right\rceil \leq |M|$.

Therefore, $|T| \leq |M|^2$.

It follows that $|M| \geq \sqrt{|T|}$.

9. [8 points]: Assuming the aggregate phase does only one pass over the data, and that $fname$ = 'AVG', describe briefly what should go in the body of the for loop (in place of "Your code goes here"). You may write pseudocode or a few short sentences. Suppose the system provides a function `emit(`*group*`,`*val*`)` to output the value of a group.

(Write your answer in the space below.)

All tuples with the same $gbyf$ will end up in the same partition, but there may (very likely will) be multiple $gbyf$ values per partition.

The basic algorithm should be something like:

```
H = new HashTable // stores (gbyf, <sum,cnt>) pairs
for tuple t in f:
  if ((<sum,cnt> = H.get(t.gbyf)) == null)
      <sum,cnt> = <0,0>
  sum = sum + t.aggf
  cnt = cnt + 1
  T.put(t.gbyf,<sum,cnt>)
for (group, <sum,cnt>) in T:
   emit(group,sum/cnt)
```

Any in-memory aggregation would work here, so if you prefer to sort the values in memory and then bucket them as is done in the next question, that is also an acceptable answer.

Name:

10. [6 points]: Describe an alternative to this hashing-based algorithm. Your answer shouldn't require more than a sentence or two.

(Write your answer in the space below.)

Remember that the tuples do not fit in memory.

One solution would be to use an external memory sort on the table, where the sort key for each tuple is the gbyf field.

After sorting, sequentially scan the sorted file's tuples, keeping a running average for each gbyf grouping (they will appear in contiguous chunks in the file). The average should be calculated on each aggf field.

Other solutions included using external memory indices, such as B+Trees, on gbyf to group the tuples on which to calculate an average.

Name:

V Cost Estimation

Suppose you are given a database with the following tables and sizes, and that each data page holds 100 tuples, that both leaf and non-leaf B+Tree pages are dense-packed and hold 100 keys, and that you have 102 pages of memory. Assume that the buffer pool is managed as described in the DBMIN Paper ("An Evaluation of Buffer Management Strategies for Relational Database Systems.", VLDB 1985.)

Table	Size, in pages
T_1	100
T_2	1000
T_3	5000

11. [15 points]: Estimate the *minimum* number of I/Os required for the following join operations. Ignore the difference between random and sequential I/O. Assume that B+Trees store pointers to records in heap files in their leaves (i.e., B+Tree pages only store keys, not tuples.)

(Write your answers in the spaces below. Justify each answer with a short analytical argument.)

- Nested loops join between T_1 and T_2, no indices.

  T1 fits in memory. Put it as the inner and read all of T1 once. Then scan through T2 as the outer. Total: $|T1| + |T2| = 1100$ I/Os.

- Grace hash join between T_2 and T_3, no indices.

  Grace join hashes all of T2, T3 in one pass, outputting records as it goes. It then scans through the hashed output to do the join. Therefore it reads all tuples twice and writes them once for a total cost of: $3(|T2| + |T3|) = 18000$ I/Os.

- Index nested loops join between a foreign key of T_2 and the primary key of T_3, with a B+Tree index on T_3 and no index on T_2.

  Put T2 as the outer and T3 as the inner. Assume the matching tuples are always on a single B+Tree page and that selectivity is 1 due to foreign-key primary-key join. Cache the upper levels of the B+Tree. T3 is 500,000 tuples, so it needs 3 levels (top level = 100 pointers, second level = $100^2 = 10,000$, third level $100^3 = 1,000,000$ pointers). Cache the root plus all but one of the level 2 pages (100 pages). Read all of T2 once (one page at a time). For each tuple in T2, we do a lookup in the B+Tree and read one B+Tree leaf page (at level 3), then follow the pointer to fetch the actual tuple from the heap file (using the other page of memory).

  Total cost is: $1000(|T2|) + 100(top\ of\ B+Tree) + \{T2\} \times No.BTree\ lookups$

  For $99/100$ of the B+Tree lookups, two levels will be cached, so $\{T2\} \times No.BTree\ lookups$ is:

  $99/100 * (2 \times |T2|) = 99/100 * 2 \times 100000 = 198000$ pages.

  For $1/100$ of the B+Tree lookups, only the root level will be cached:

  $1/100 * (3 \times |T2|) = 1/100 * 3 \times 100000 = 3000$ pages.

  So the total is :

  $1000 + 100 + 198000 + 3000 = 202100\ I/Os$

  We were flexible with the exact calculation here due to the complexity introduced by not being able to completely cache both levels of the B+Tree.

Name:

VI ARIES with CLRs

Suppose you are given the following log file.

LSN	TID	PrevLsn	Type	Data
1	1	-	SOT	-
2	1	1	UP	A
3	1	2	UP	B
4	2	-	SOT	-
5	2	4	UP	C
6	-	-	CP	dirty, trans
7	3	-	SOT	-
8	2	5	UP	D
9	3	7	UP	E
10	1	3	COMMIT	-
11	2	8	UP	B
12	2	8	CLR	B
13	3	7	CLR	E

12. [2 points]: After recovery, which transactions will be committed and which will be aborted?
(Write your answers in the spaces below)

Committed:_____

Aborted:_____

T1 commits, while T2 and T3 abort, since T1 is the only transaction that has a COMMIT record in the log.

13. [4 points]: Suppose the dirty page table in the CP record has only page A in it. At what LSN will the REDO pass of recovery begin?
(Write your answer in the space below)

LSN:_____

The LSN that the REDO pass will start at is 2.

The LSN selected is the earliest recoveryLSN in the dirty page table for any dirty page. Since only page A is in the dirty page table, and the first log record in which it was modified (its recoveryLSN) is 2, the REDO pass will begin at LSN 2. Pages B and C might have been stolen (STEAL) by some background flush process before the checkpoint was written out, and so they do not appear in the dirty page table.

Name:

14. [4 points]: Again, suppose the dirty page table in the CP record has only page A in it. What pages may be written during the REDO pass of recovery?

(Write your answer in the space below)

Pages:_____

Pages A, B, D, E. REDO will start at LSN 2, and re-apply any UP record in the log. Since B and C are not in the dirty page table, any change to them before the checkpoint must already be on disk, so they are not written by REDO before the checkpoint. Since B is updated after the checkpoint, it will be written at that point.

15. [4 points]: Once again, suppose the dirty page table in the CP record has only page A in it. What pages may be written during the UNDO pass of recovery?

(Write your answer in the space below)

Pages:_____

Pages C and D. The UNDO stage starts at the last LSN for the last transaction to be aborted, and proceeds backward through the log. LSN 13 is read first, but since it is a CLR (previous recovery handled it), E was correctly updated on disk, and is not rewritten. LSN 12 is also a CLR, so we skip the update to B for the same reason. Following LSN 13's prevLSN pointer, we see that LSN 7 was the start of transaction 3, so we are done undoing transaction 3. Following LSN 12's prevLSN pointer leads us to LSN 8, and so we undo the update to D (overwriting D). Following LSN 8's prevLSN pointer to LSN 5, we again undo the update to C (overwriting C). Following LSN 5's prevLSN, we go to LSN 4, which is the start of transaction 2, and thus the end of the UNDO pass.

VII Snapshot Isolation

Oracle and Postgres both use a form of transaction isolation called snapshot isolation. One possible implementation of snapshot isolation is as follows:

- Every object (e.g., tuple or page) in the database has a timestamp; multiple copies ("versions") of objects with old timestamps are kept until no transaction needs to read them again. (For this question, you don't need to worry about how such old versions are maintained or discarded.)
- When a transaction begins, the system records the transaction's start time stamp, ts_s. Timestamps are monotonically increasing, such that no two transactions have the same timestamp value.
- When a transaction T writes an object O, it adds the new version of the O to T's *local write set*. Versions in the local write set are not read by other transactions until after T has committed.
- When a transaction T reads an object O, it reads the most recent committed version with timestamp $\leq ts_s$, reading O from T's own local write set if O has been previously written by T.
- When a transaction T commits, a new timestamp, ts_c is taken. For every object O in T's local write set, if the most recent version of O in the database has timestamp $\leq ts_s$, then O is written into the database with timestamp ts_c. Otherwise, T aborts. Only one transaction commits at a time.

For example, consider the following schedule:

Initial database: objects A and B, both version 0 (denoted A_0 and B_0)

$T1(ts_s = 1)$	$T2(ts_s = 2)$
$Read(A_0)$	
$Write(A)$	
	$Read(A_0)$
	$Write(A)$
commit ($ts_c = 3$)	
install A_3	
	attempt to commit with $ts_c = 4$,
	but abort, because last version of A (3) $> ts_s = 2$

Here, $T2$ aborts because it tried to write A concurrently with T1.

16. [10 points]: Is snapshot isolation conflict serializable? If yes, state briefly why. If not, give an example of a non-serializable schedule.

(Write your answer in the space below.)

No. Snapshot isolation ignores read/write conflicts. Consider T1: (R_A, W_B) and T2: (R_B, W_A). These execute in the following order:

R_{A0} (T1), R_{B0} (T2), W_{B1} (T1), W_{A2} (T2)

Under snapshot isolation, this order is acceptable and both transactions can commit. However, this is neither serial orders T1, T2 (where T2 would R_{B1}) or T2, T1 (where T1 would R_{A2}).

Name:

17. [5 points]: Oracle claims that snapshot isolation is much faster than traditional concurrency control. Why?

(Write your answer in the space below.)

Writes never block readers. For example, if a long running transaction has written value A, with traditional concurrency control any transaction that wants to read A will need to wait to acquire a lock on A. With snapshot isolation, the reading transactions will read the previous version and continue.

This is similar to optimistic concurrency control in that there are no locks and transactions do not block each other. However, with snapshot isolation, a transaction sees a consistent snapshot of the database, determined when it begins. With optimistic concurrency control, a transaction can see values from transactions that commit after it begins.

End of Quiz I

Name: