UNIVERSITY OF CALIFORNIA
Department of EECS, Computer Science Division

CS186 Hellerstein
Spring 2010 Final Exam

Final Exam: Introduction to Database Systems

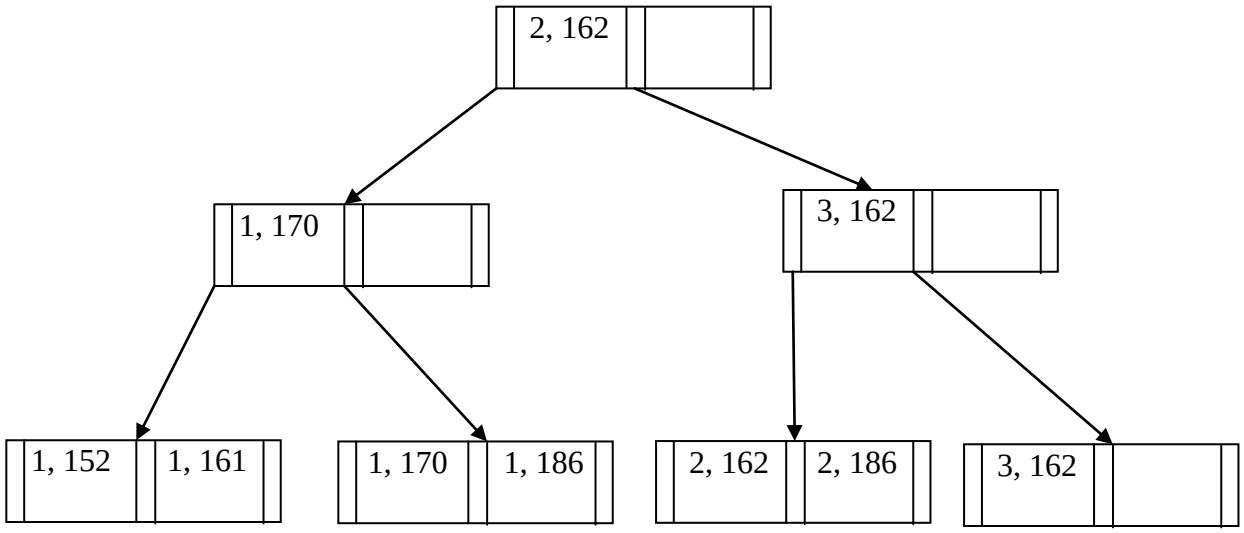
Solutions are in red. Correct answers intended to be circled are highlighted.

Q1: Tree-Structured Indexes [10 points]

Consider the instance of the Enrolled table:

Student ID	Course ID
1	186
2	186
1	161
1	170
1	152
2	162
3	162

a) Use the bulk-loading algorithm to create an "Alternative 1" B+Tree index below on (Student ID, Course ID). Assume 2 entries (3 pointers) fit per internal node, with a minimum of 1 entry (2 pointers). Assume 2 entries fit per leaf node. Fill leaf nodes to capacity. Draw your solution below. **[4 points]**



(The left internal node is not completely full because otherwise the right internal node would be underfull.)

b) Consider an "Alternative 1" B+Tree of height H where internal nodes and leaf nodes both hold R entries (internal nodes also hold R+1 pointers). All intermediate nodes (including the root) are full, and all leaf nodes are at least half-full. Given this constraint and the usual constraints of a B+Tree, assume whatever data you want in the tree for each part below. Assume that we measure height starting at 1 -- i.e. a 1-node B+Tree has height 1. **[2 points]**

i) What is the maximum number of inserts possible *before* the root splits?

$((R+1)^H - 1)(\text{floor}(R/2))$ (assuming all leaf nodes are half full)

ii) What is the minimum number of inserts that would cause the root to split?

1 (assuming insertion into a full leaf node)

c) Fill in the cost table below for "Alternative 1" ISAM and B+Tree indices: **[4 points]**

Assume each index takes P pages on disk, has height H, and fanout F at each internal node. Assume there are R tuples in the relation, and B tuples fit on a leaf (or overflow) page. In each case, assume infinite buffer pool size, but the buffer pool starts out empty. For each page that you dirty, add 1 to your I/O cost since it will eventually have to be flushed to disk. For ISAM, assume that a leaf node maintains only a pointer to the beginning of an overflow list. Given the constraints of a B+Tree/ISAM, assume whatever data you want in the tree for each case below.

	Worst-case # I/Os for Range Query	Worst-case # I/Os to Insert
ISAM	**P** *(index consists of root with a linear string of overflow pages. Need to look at all overflow pages since they're not sorted)* *or* **H + R/B** ***or*** **H + (F^H – 1) + R/B** (look at whole leaf level and all data in last leaf overflow)	**P+2** *(index consist of a root with a string of overflow pages. Need to scan til the end, and add a new overflow page in the worst case, and update the previous last overflow page with a pointer)* **H + R/B + 2**
B+Tree	**H + F^H** *(range query covers the whole table)* **H + R/B** **P** was **not** accepted here, as this would imply only 2 I/Os, given the structure of the index.	**3 H + 1** *(every node needs to split, +1 for new root. Read pages we're going to split on the way down, so we don't need to read them again.)*

Q2: Normal Forms [11 points]

Consider the "Congress" relation, and associated functional dependencies:

Congress(Bill, Title, Sponsor, Party, District, Committee, cHairperson, chAirperson_party, heaRing_time)

R → SP
SP → DCH
B → SCT
DH → A
TS → R
SPR → B
S → P

a) **[4 points]** All of the candidate keys for the relation above are listed below, possibly along with some attribute sets that are not candidate keys. Circle the attributes sets that *are* candidate keys for the relation above.

- R
- S
- B
- TS

b) Circle the constraints below (if any) that violate BCNF. **[4 points]**

1. R → SP
2. SP → DCH
3. B → SCT
4. DH → A
5. TS → R
6. SPR → B
7. S → P
8. None of the above

c) Consider the following relation and functional dependencies:

SupremeCourt(Docket, Appellant, Respondent, Oral_argument_time, oPinion_author, appoInted_by, parTy)

1. PI → T
2. RP → I
3. O → ARP
4. D → O
5. OA → D

i) Write the lossless-join decomposition of this relation into BCNF, by resolving the constraints that violate BCNF (if any) in numerical order. **[2 points]**

DAROPIT decomposes by (1) into DAROPI and PIT, since PI does not determine all attributes.
DAROPI decomposes by (2) into DAROP and RPI, since RP does not determine all attributes.
O determines ARP, and OA determines D, so O is a superkey of DAROP.
D determines O, and O is a superkey of DAROP, so D is a superkey of DAROP.
Final lossless-join decomposition: DAROP, RPI, PIT

ii) Is this decomposition dependency-preserving? **[1 point]**

Yes (for every constraint, all columns in the constraint are in a single table in the decomposition)

d) Assume that you considering a new normal form TANF (Totally Awesome Normal Form). A relational schema R satisfies TANF if, for every functional dependency X → Y, one of the following is true:

i) X → Y is a trivial FD
ii) X is a candidate key for R

Assume you decompose a relation R into TANF in the same way you decompose a relation into BCNF. Does this decomposition for TANF always have the lossless-join property? If yes, provide a 2.5-line argument. If no, provide a counterexample involving at most two FDs. Longer answers will receive no credit. **[3 points]**

If YES, write argument here: **TANF is a subset of BCNF, and BCNF has the lossless-join property.**

If NO, write counterexample here:

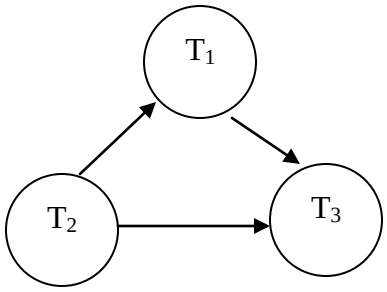
Q3: Concurrency [10 points]

Consider the following schedule of accesses by three transactions. The labels R and W indicate reads and writes, and the labels A, B, and C indicate distinct elements of data.

There are many correct answers to this question – as long as each piece of data is locked before used, each row contains at most one item, and no transaction locks data after it begins unlocking, the answer is correct.

Time	T_1	T_2	T_3
1		L(A)	
2		R(A)	
3	L(C)		
4	R(C)		
5		L(B)	
6		R(B)	
7			
8		W(B)	
9		U(B)	
10		U(A)	
11			L(B)
12			R(B)
13	L(A)		
14	R(A)		
15	U(C)		
16			L(C)
17			R(C)
18			
19			W(C)
20			U(C)
21	W(A)		
22	U(A)		
23			U(B)

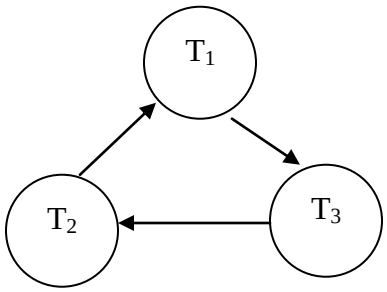
(a) **[2 points]** Recall the definition of a precedence graph: "A precedence graph has a node for each committed transaction, and an arc from *Ti* to *Tj* if an action of *Ti* precedes and conflicts with one of *Tj*'s actions." Draw a precedence graph for the schedule on the previous page.



(b) **[2 points]** Is the schedule on the previous page conflict-serializable? If so, what order should the transactions be executed in to produce a conflict-equivalent serial schedule?

The precedence graph contains no cycles, so the schedule is conflict-serializable. The only possible conflict-equivalent serial schedule is T2, T1, T3.

(c) **[1 point]** Suppose instead of reading B at time 12, transaction 3 reads B at time 7. Draw a precedence graph for this modified schedule.



(d) **[1 point]** Is the schedule of part (c) conflict-serializable? If so, what order should the transactions be executed in to produce a conflict-equivalent serial schedule?

The precedence graph contains a cycle, so the schedule is not conflict-serializable.

(e) **[4 points]** Add lock/unlock actions into the schedule on the previous page in a way compliant with (non-strict) two-phase locking. Use L(X) to lock a data element X, and U(X) to unlock it. At most one box on each row should contain an action, and it may contain only one action. You should only use exclusive locks, not shared (read) locks. No locks should remain held at the end of the schedule.

See previous page.

Q4: Logging and recovery [11 points]

Your database server has just crashed due to a power outage. You boot it back up, find the following log and checkpoint information on disk, and begin the recovery process. Assume we use a **STEAL/NO FORCE** recovery policy.

LSN	Record	prevLSN
30	update: T3 writes P5	null
40	update: T4 writes P1	null
50	update: T4 writes P5	40
60	update: T2 writes P5	null
70	update: T1 writes P2	null
80	Begin Checkpoint	-
90	update: T1 writes P3	70
100	End Checkpoint	-
110	update: T2 writes P3	60
120	T2 commit	110
130	update: T4 writes P1	50
140	T2 end	120
150	T4 abort	130
160	update: T5 writes P2	Null
180	CLR: undo T4 LSN 130	150

Transaction table at time of checkpoint

Transaction ID	lastLSN	Status
T1	70	Running
T2	60	Running
T3	30	Running
T4	50	Running

Dirty page table at time of checkpoint

Page ID	recLSN
P5	50
P1	40

(a) [3 points] The log record at LSN 60 says that transaction 2 updated page 5. Was this update to page 5 successfully written to disk? The log record at LSN 70 says that transaction 1 updated page 2. Was this update to page 2 successfully written to disk? Explain briefly in both cases.

The update at LSN 60 *may* have been written to disk; the log entry was flushed before the write itself. It was not yet flushed at the time of the checkpoint, but may have been flushed later.

The update at LSN 70 *was* flushed to disk. We know this because it's not in the dirty page table at the time of the checkpoint.

(b) [4 points] At the end of the Analysis phase, what transactions will be in the transaction table, and with what lastLSN and Status values? What pages will be in the dirty page table, and with what recLSN values?

Transaction ID	lastLSN	Status
T1	90	Running
T3	30	Running
T4	180	Aborting
T5	160	Running

Page ID	recLSN
P1	40
P2	160
P3	90
P5	50

(c) [4 points] At which LSN in the log should redo begin? Which log records will be redone (list their LSNs)? All other log records will be skipped.

Redo should begin at LSN 40, the smallest of the recLSNs in the dirty page table. The following log records should be redone:

40, 50, 60, [80], 90, [100], 110, [120], 130, [140], [150], 160, 180

30 is skipped because it precedes LSN 40. 70 is skipped because P2.recLSN = 160 > 70. Entries that are not updates are skipped. The CLR record is *not* skipped, nor is the LSN that it undoes.

Q5: Search and Query Processing [10 points]

You are consulting on the design of a new search engine. The company building it wants to use SQL on top of a DBMS. (You tell them that using a DBMS is not the best approach for high-performance text search. They tell you it is a non-negotiable design decision. You nod reasonably; this is not your first time working with an irrational customer!)

a) **[4 points]** The company has prototyped basic Boolean search on a small test data set. They are storing the files in a single table of the form

```
Files(docID integer, content text, PRIMARY KEY (docID)).
```

And they have a table of StopWords as well.

Here's their query template for a 2-keyword search ($1 and $2 are replaced with keywords at runtime):

```
SELECT DISTINCT A.docID
  FROM Files A, Files B, StopWords S
 WHERE A.docID = B.docID
   AND A.content LIKE %$1%
   AND B.content LIKE %$2%
   AND $1 <> S.word
   AND $2 <> S.word;
```

For each of the following comments, answer True or False, and explain your answer in the space provided (DO NOT use more space!):

i. This query is exponential in the number of File tuples, so it will get exponentially slower as they add files to their corpus.

False. With hash join should be at worst O(nlogn) growth.

ii. The self-join in this query is useless.

True. A self-join on the key just matches files with themselves. Could drop Files B and the join clause, and test all predicates on Files A.

iii. This query will produce no output for the keyword $1 = "the" as long as it was inserted into the StopWords table.

False. The inequality join condition will be satisfied by all other words in Stopwords.

iv. The query optimizer may produce ridiculously bad join orders.

True. It is very hard for a query optimizer to predict the reduction factor of a predicate with LIKE in it, and this can vary widely. Bad join order (say A join B rather than B join A) can cost a lot.

==========

If your answer continues below here it is TOO LONG.

b) **[4 points]** The company likes your idea of using inverted indexes. They propose to use the scheme we described in class: build an `InvertedFile` relation in the DBMS with an "Alternative 3" B-tree index on the term column. The data entries in the leaves point to `RecordIds` of the `InvertedFile` heap file in the database.

You explain to them that their DBMS *will not ensure* that the "Alternative 3" entries are sorted by `RecordId`. So the optimizer *will not be able* to choose the "standard" query plan from class using merge join. They don't see any problem with that.

To demonstrate, you show the Boolean query "Miley AND antidisestablishmentarianism". The data entry (postings list) for "Miley" takes 350MB(42.9 million results on Google), and the one for "antidisestablishmentarianism" takes 5MB (81,200 results on Google). They have 10MB of buffer space to run this query.

Assume the optimizer does a good job choosing among the various join algorithms and access methods we learned in class. Draw the query plan it would choose, and write down the total I/O cost including index access and join costs (but not the cost of writing out the answer).

Picture: HashJoin(IX-SCAN_anti(InvertedFile), IX-SCAN_miley(InvertedFile)).

First, hash the postings for "antidisestablishmentarianism"; it fits in memory. Then simply stream the results of the "Miley" postings out of the index and look each up in the main-memory hashtable. Block Nested Loop gives a similar analysis.

2*IHeight + (5MB + 350 MB)/ B where B is the number of MBytes/Block.

c) **[2 points]** What would the I/O cost have been using the scheme described in class: i.e. postings lists guaranteed to be sorted by docID, and simple merge join?

2*IHeight + (5MB + 350 MB)/ B where B is the number of MBytes/Block.

Q6: A Little SQL [4 points]

The questions on this page refer the the relation defined by this statement:

```
CREATE TABLE Students(id integer, gpa float, name text,
                      address text, gender char,
                      PRIMARY KEY (id));
```

a. **[1 point]** Are the two queries below equivalent ? That is, do they return the same answer on any database instance? Answer True of False; no explanation required.

```
SELECT MAX(S.id) FROM Students S;

SELECT S.id FROM Students S
 WHERE S.id >= ALL (SELECT S2.id FROM Students S2);
```

YES

b. **[1 point]** Among the 3 queries below, some or all are equivalent. Circle the ones that are equivalent.

```
SELECT MAX(S.gpa) FROM Students S;

SELECT S.gpa FROM Students S
 WHERE S.gpa >= ALL (SELECT S2.gpa FROM Students S2);

SELECT S.gpa FROM Students S
GROUP BY S.gpa
HAVING S.gpa >= ALL (SELECT S2.gpa FROM Students S2
                     WHERE S2.gpa > S.gpa);
```

c. **[1 point]** Consider the following query and the table of data to the right:

```
SELECT S.id FROM Students S
 WHERE S.gpa > 3.3
   AND S.id > 120;
```

id	gpa	name	address	gender
123	null	Joe	38 Maple	M
124	3.2	Hui	64 Vine	F
127	3.9	Celia	21 Elm	F
111	3.2	Hector	11 Oak	M

How many rows should be in the output?

1

d. **[1 point]** Using the same data from the table in part (c), how many rows should be in the output of the following query?

```
SELECT S.id FROM Students S
 WHERE S.gpa > 3.3 OR S.gender = 'M';
```

3

Q7: More SQL [8 points]

Consider this old chestnut: the *Stable Marriage Problem*, described on its Wikipedia page as follows.

> Given *n* men and *n* women, where each person has ranked all members of the opposite sex with a unique number between 1 and *n* in order of preference, marry the men and women off such that there are no two people of opposite sex who would both rather have each other than their current partners. If there are no such people, all the marriages are "stable".

The (arguably old-fashioned) algorithm at Wikipedia has the following pseudocode:

```
 1 function stableMatching {
 2     Initialize all m ∈ M and w ∈ W to free
 3     while ∃ free man m who still has a woman w to propose to {
 4        w = m's best ranked such woman who he has not proposed to yet
 5        if w is free
 6          (m, w) become engaged
 7        else some pair (m', w) already exists
 8          if w prefers m to m'
 9            (m, w) become engaged
10            m' becomes free
11          else
12            (m', w) remain engaged
13     }
14 }
```

We will implement a batch-oriented scalable version of this algorithm in SQL using the following schema. The first two tables are the input to the algorithm, the last four are used in the implementation.

```
-- for each male, store a pref for each female -- the lower the better (1 is best, 2 is 2nd-best, etc). status is either
-- 'f' for free, or 'e' for engaged, and should preserve the FD mID->status.
CREATE TABLE M (mID integer, fID integer, pref integer, status char,
                PRIMARY KEY (mID, fID));

-- similarly for each female, store a pref for each male, but here preserve fID -> status.
CREATE TABLE F (fID integer, mID integer, pref integer, status char,
                PRIMARY KEY (fID, mID));

-- keep track of prior proposals (for tests in lines 3-4)
CREATE TABLE proposals (mID integer, fID integer);

-- keep track of engagements
CREATE TABLE engaged (mID integer, fID integer);

-- each round we will have a set of new proposals to consider
CREATE TABLE newproposals (mID integer, fID integer);

-- some rounds we may find engaged women who would prefer to upgrade to a new proposal (lines 8-9)
CREATE TABLE upgrades (newMan integer, fID integer, oldMan integer);
```

a. **[1 point]** Translate line 2 of the pseudocode into SQL over the schema above.

```
UPDATE M SET status = 'f';
UPDATE F SET status = 'f';
```

b. **[3 points]** Fill in the following SQL, for generating a set of all (`mID, fID`) pairs corresponding to a batch of (*m,w*) pairs from lines 3 and 4.

```
DELETE FROM newproposals;

INSERT INTO newproposals
  SELECT MIN(M.mID), M.fID
    FROM M
   WHERE M.status = 'f'
     AND M.pref =

          (SELECT MIN(M2.pref)
             FROM M AS M2
            WHERE M2.mID = M.mID
              AND NOT EXISTS (SELECT *
                                FROM proposals AS p


                               WHERE p.mID=M2.mID


                               AND p.fID = M2.fID)
          )
  GROUP BY M.fID;
```

c. **[2 points]** A slightly simpler version of the previous query would omit the `GROUP BY` clause, and use `M.mID` rather than `MIN(M.mID)` in the `SELECT` list. What problem could arise in this simpler version of the query?

Multiple men would propose to the same woman "at once".

d. **[2 points]** Fill in the query below corresponding to lines 5 and 6 of the pseudocode:

```
INSERT INTO engaged
SELECT DISTINCT P.mID, P.fID
  FROM newproposals AS P, F



 WHERE P.fID = F.fID



 AND F.status = ‘f’
```

THE END!

For your entertainment, here is some SQL that corresponds to lines 7-8 of the pseudocode (lines 11-12 are a no-op).

```
DELETE FROM upgrades;

INSERT INTO upgrades
SELECT p.mID AS newMan, p.fID, engF.mID AS oldMan
  FROM newproposals AS P, F as newF, F as engF, engaged AS E
 WHERE P.fID = newF.fID
   AND P.mID = newF.mID
   AND newF.fID = engF.fID
   AND engF.fID = E.fID
   AND engF.mID = E.mID
   AND newF.status = 'e'
   AND newF.pref < engF.pref;

DELETE FROM engaged
 WHERE (mID, fID) IN (SELECT oldMan, fID FROM upgrades);

UPDATE M SET status = 'f'
 WHERE mID IN (SELECT oldMan FROM upgrades);

INSERT INTO engaged
SELECT newMan, fID
  FROM upgrades;

UPDATE M SET status = 'e'
 WHERE mID IN (SELECT mID FROM engaged);

UPDATE F SET status = 'e'
 WHERE fID IN (SELECT fID FROM engaged);
```