CSE 414 Final Exam Sp13 Sample Solution

Question 1. (28 points) SQL. The following database contains information about actors, plays, and roles performed.

Actor(actor_id, name, year_born)
Play(play_id, title, author, year_written)
Role(actor_id, character_name, play_id)

Where:

- Actor is a table of actors, their names, and the year they were born. Each actor has a unique actor_id, which is a key.
- Play is a table of plays, giving the title, author, and year written for each play. Each play has a unique play_id, which is a key.
- Role records which actors have performed which roles (characters) in which plays. Attributes actor_id and play_id are foreign keys to Actor and Play respectively. All three attributes make up the key since it is possible for a single actor to play more than one character in the same play.

(a) (8 points) Write the SQL statements that define the relational schema (tables) for this database. Assume that actor_id, play_id, year_born, and year_written are all integers, and that name, title, author, and character_name are strings. Be sure to define appropriate keys and foreign key constraints.

```
CREATE TABLE Actor (
        actor_id INTEGER PRIMARY KEY,
        name VARCHAR(100)                -- any reasonable SQL character string type is ok
        year_born INTEGER
);

CREATE TABLE Play (
        play_id INTEGER PRIMARY KEY,
        title VARCHAR(100),
        author VARCHAR(100),
        year_written INTEGER
);

CREATE TABLE Role (
        actor_id INTEGER REFERENCES Actor(actor_id),
        character_name VARCHAR(100),
        play_id INTEGER REFERENCES Play(play_id),
        PRIMARY KEY(actor_id, character_name, play_id)
);
```

Actor(actor_id, name, year_born)
Play(play_id, title, author, year_written)
Role(actor_id, character_name, play_id)

Question 1. (cont) (b) (6 points) Write a SQL query that returns the number of actors who have performed in three or more different plays written by the author "August Wilson"

```
SELECT count(r.actor_id)
FROM Role r, Play p
WHERE p.author = 'August Wilson' AND p.play_id = r.play_id
GROUP BY r.actor_id
HAVING count(DISTINCT r.play_id) > 2
```

(c) (6 points) Write a SQL query that returns the names of all actors who have performed some play by the author "Chekhov" and have never performed in any play written by author "Shakespeare". The list should not contain duplicates but does not need to be ordered.

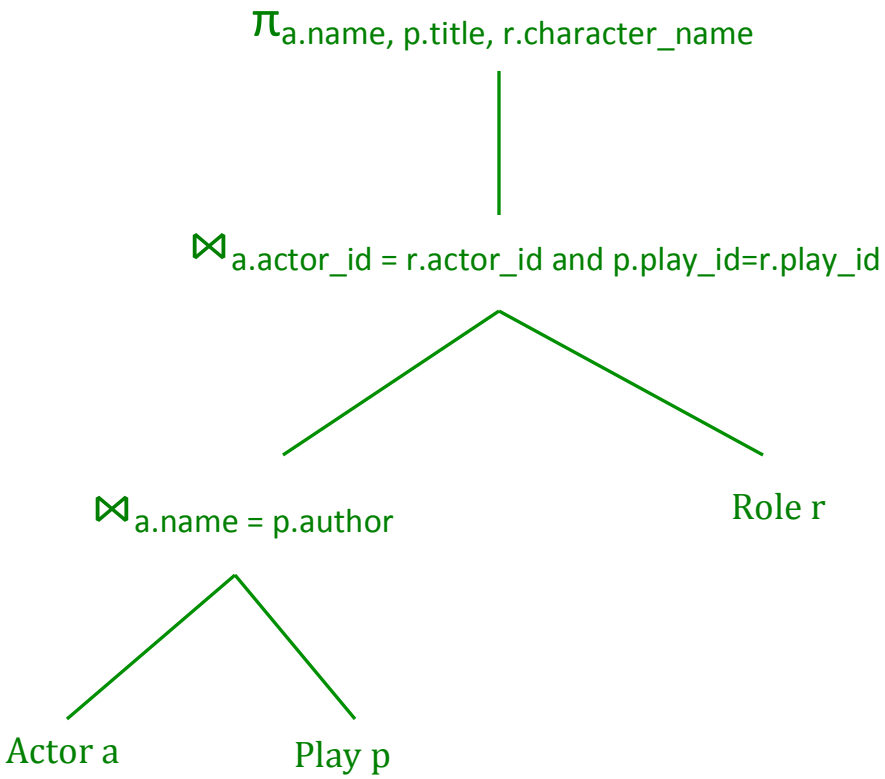
```
SELECT DISTINCT a.name
FROM Actor a, Play p, Role r
WHERE p.author = 'Chekhov' AND p.play_id = r.play_id AND r.actor_id = a.actor_id AND
      a.actor_id NOT IN (SELECT r2.actor_id
                         FROM Role r2, Play p2
                         WHERE p2.author = 'Shakespeare' AND p2.play_id = r2.play_id)
```

Actor(actor_id, name, year_born)
Play(play_id, title, author, year_written)
Role(actor_id, character_name, play_id)

Question 1. (cont.) (d) (8 points) The following query returns information about all persons who have acted in a play that they have written:

```
SELECT a.name, p.title, r.character_name
FROM Actor a, Play p, Role r
Where a.name = p.author AND a.actor_id = r.actor_id AND p.play_id = r.play_id
```

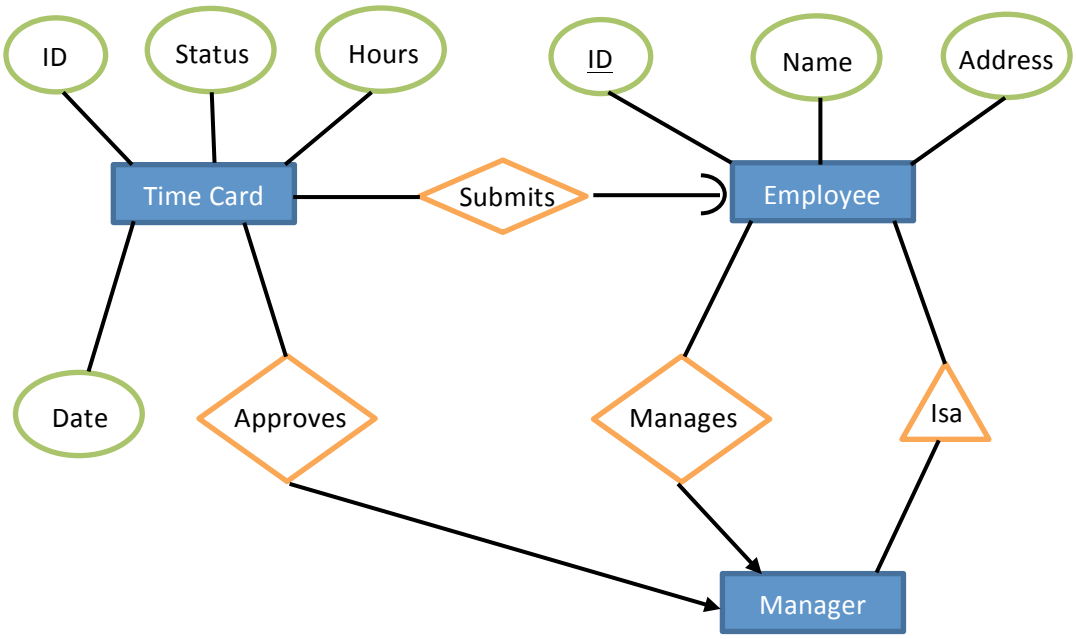
Give a relational algebra query plan drawn as a tree that correctly computes this query.



Question 2. (14 points) Bert the Payroll Guy is about to retire after 40 years and it's time to replace his manual time card system with some sort of computerized database. You have been asked to come up with the database design. As best we can tell, the time card system has the following properties:

- A *timecard* contains hours worked and date submitted
- Each *timecard* is associated with exactly one *employee*
- Each *timecard* has a unique id
- Each *timecard* has a status: approved, not approved, or pending (not examined yet)
- Each *employee* has a name, address, and a unique id
- Each *employee* submits a time card every pay period. i.e. In 1 year, they will submit multiple time cards
- Each *employee* is associated with exactly one *manager*
- Each *manager* is also an *employee*
- Each *manager* is in charge of one or more employees
- Each *manager* approves time cards for one or more employees

Draw an ER diagram that captures this information.



Question 3. (15 points) Consider the relation with schema R(A,B,C,D,E,F) and the following functional dependencies (FDs):

A → BC D → AF

(a) (7 points) What are the keys and superkeys of this relation? (Recall that a key is a minimal superkey.) Justify your answer(s) by showing the closures that are involved, and be sure to clearly label which superkeys are also keys.

The minimal key is DE since D+=ABCDF and DE+ is ABCDEF

Other superkeys are:

ADE	**ABDE**	**CDEF**
ABCDEF	**ACDE**	**ABCDE**
BDE	**ADEF**	**BCDEF**
CDE	**BCDE**	
DEF	**BDEF**	

(In grading we gave some slack if not every key was listed since the list is long and it's easy to miss one under time pressure. But it was important to identify the key (DE).

(b) (8 points) Is relation R in BCNF? If it is, explain why it is. If it is not, explain why not and give a decomposition of R into a collection of relations that are in BCNF.

No. R is in BCNF if, for every non-trivial FD X→Y, we have that X+ = all keys. Or, in other words FD X→Y violates BCNF if X ≠ X+ ≠ all attributes of R.

In this case, A→BC violates BCNF since A+ = ABC ≠ ABCDEF. So we split R into R1(ABC) and R2(ADEF).

The only non-trivial FD in R1 is A→BC, and A+ = ABC, so R1 is in BCNF.

R2 has a non-trivial dependency D→AF that violates BCNF because D+ = ADF ≠ ADEF. So we split R2 into R21(DAF) and R22(DE). Both of these are in BCNF since they have no non-trivial dependencies that are not superkeys.

It would also be possible to use the FD D+ = ABCDF ≠ ABCDEF as the initial dependency for the split. In that case we would get different intermediate steps in the decomposition, but the final results would be the same.

Question 4. (12 points) Cost estimation. Suppose B(R) = 5000, T(R) = 250000, B(S) = 1000, T(S) = 4000, and M = 1200.

(a) (6 points) What is the expected cost of performing a nested loop join between R and S? If you have any choices in how to carry out the join operation, pick the strategy with the lowest cost – but be sure it is a nested loop join. Briefly explain your strategy (pseudo-code would be helpful) and show enough work so we can understand how you arrived at your answer.

The simple cost estimation we presented in lecture is that if we read a block of each relation at a time, the cost of joining R and S is B(S) + B(R)B(S). (We can use either R or S as the outer relationship since join is a commutative operation and, in this case, since S occupies fewer blocks we get a lower cost that way.)

```
for each block s in S do
  for each block r in R do
    for each pair of tuples in r and s, if the pair of tuples join, add them to the result.
```

In this case we would need 1000+5000*1000 or 5,001,000 disk read operations.

That analysis was enough to get full credit. However, if we take advantage of the memory that is available, we can reduce the cost to approximately B(S) + B(R)B(S)/(M-1). The idea is to read as many as M-1 blocks of one relation into memory, then join each block from the other relation to all of the blocks in memory before reading more of the first relation. So we have something like this:

```
repeat
  read up to M-1 blocks of S into memory
  for each block r in R
    for each tuple in r, if it joins with any of the tuples of S currently in memory,
          add those pairs of tuples to the result.
```

Since all of relation S will fit in main memory, we can read all of it at once, then join blocks of R to it. Since we only need one trip around the outer loop, the total cost is approximately B(S) + B(R)B(S)/M or 1000 + 5000*1000/1000 or 6000.

(b) (6 points) Is it possible to perform a hash join between R and S, given the above values for B(R), B(S), and M? If so, describe the high-level algorithm steps and compute the cost of the join. If it is not possible to perform a hash join under these circumstances, explain why not.

Yes. There is enough main memory to hold all of S. So we read S into a hash table at cost B(S) = 1000, then read the blocks of R at cost B(R) = 5000 and join the tuples to S using a hash lookup instead of sequentially searching the blocks of S in main memory. The total cost is again 1000+5000 = 6000.

Question 5. (12 points) Consider an XML document containing information about job postings and job applications. The postings are grouped by company, and applications are listed under postings. An application contains only the applicant's name. For example:

```
<jobs>
  <company>
    <name> MicroScience Corp. </name>
    <posting>
      <jobtitle> sales rep </jobtitle>
      <salary> 30000 </salary>
      <application> Mark </application>
      <application> Lucy </application>
      <application> Frank </application>
    </posting>
    <posting>
      <jobtitle> technology consultant </jobtitle>
      <salary> 80000 </salary>
      <application> Lucy </application>
      <application> Fred </application>
      <application> Mark </application>
      <application> Betty </application>
    </posting>
  </company>
  <company>
    <name> MacroConsulting Inc. </name>
    <posting>
      <jobtitle> technology consultant </jobtitle>
      <salary> 40000 </salary> <application> Frank </application>
    </posting>
    <posting>
      <jobtitle> debugger </jobtitle>
      <salary> 20000 </salary>
    </posting>
    <posting>
      <jobtitle> programmer analyst </jobtitle>
      <salary> 35000 </salary>
      <application> Lucy </application>
      <application> Mark </application>
    </posting>
  </company>
</jobs>
```

Answer questions about this document on the next page. You can remove this page from the exam if you wish.

Question 5. (cont) (a) (6 points) For each of the XPath expressions below, indicate how many answers it will return on the example XML document on the previous page. For example, if the XPath expression is

```
/jobs/company[name/text()='MacroConsulting Inc.']/posting
```

then your answer should be 3. Your answer to each part should be only a number.

(i) `//salary`

5

(ii) `/jobs/company/posting[salary/text()>50000]//application`

4

(iii) `/jobs/company[posting/salary/text()>50000]//application`

7

(b) (6 points) Write an XQuery expression that returns the names of all applicants who have submitted applications to at least two companies. Your query should return a list of <name> elements inside a root element <applicants>, and each element should be included only once. For example, if your query were run on the XML document shown above, it should return

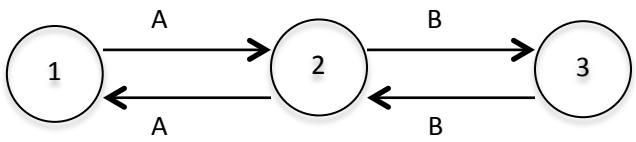
```
<applicants>
<name>Mark</name>
<name>Lucy </name>
<name>Frank </name>
</applicants>
```

```
<applicant>
{
for $all in doc("test.xml")/jobs
for $applicants in distinct-values($all//application/text())
let $company := $all/company[posting/application/text()=$applicants/name/text()]
where count(distinct-values ($company)) >= 2
return <name> { $applicants } </name>
}
</applicant>
```

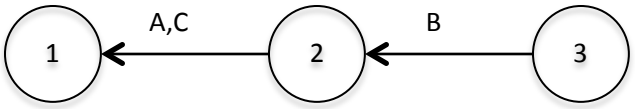
Question 6. (12 points, 6 each) Serializibility. For each of the following transaction schedules, draw the precedence (conflict) graph and decide if the schedule is conflict-serializable. If the schedule is conflict-serializable, give an equivalent serial schedule (you just need to list the order of transactions, not all the individual read-write operations, although you can give the full schedule if it is helpful). If the schedule is not conflict-serializable, explain why not.

(a) r1(A); r2(A); w2(A); r3(B); r2(B); w3(B); w2(B); w1(A)

This is not conflict-serializable. T1 and T2 have RW conflicts in both directions on A, and T2 and T3 similarly have RW conflicts on B. There are cycles in the graph so it is not conflict-serializable.



(b) r1(A); r2(A); r3(B); w1(A); r2(C); r2(B); w2(B); w1(C)



This schedule is conflict-serializable. T3 must occur before T2 because of a RW conflict on B and T2 must occur before T1 because of RW conflicts on A and C. Since there are no cycles, T3, T2, T1 is an equivalent serial schedule.

Question 7. (12 points) Locking. In an attempt to guarantee conflict-serializable execution of transactions we introduced the notion of locks. The idea is that transaction *i* should perform a lock operation Li(A) on element A before reading or writing that element, then perform an unlock operation Ui(A) when it is done. But we discovered that this rule was not sufficient to guarantee a serializable schedule and we needed to use the more complex two-phase locking algorithm (2PL) instead.

(a) (6 points) Give an example showing how the use of simple locks without 2PL is not enough to guarantee conflict serializibility.

This example from lecture, or something similar, captures the problem.

T1: L1(A), r1(A), w1(A), U1(A)

T2: L2(A), r2(A), w2(A), U2(A), L2(B), r2(B), w2(B), U2(B)

Now if, after T2 finishes, T1 does L1(B), r1(B), w1(B), U1(B), then the resulting schedule is not conflict serializable since neither serial order T1 before T2 nor T2 before T1 is equivalent to that schedule.

(b) (6 points) What is two phase locking (2PL) and how does it solve the problem(s) you identified in part (a) above?

2PL requires that all lock requests issued by a transaction must precede any unlock requests. Although it is possible for transactions to block (or deadlock) with 2PL, if the transaction finishes 2PL guarantees serializibility because no other transaction can interfere with some of the items being used by the transaction or read partially updated results while it is working. (There are more formal ways to state it, but that is the general idea.)

Question 8. (15 points) Map-Reduce. We have some very large log files storing information about the traffic on a computer messaging service. The entries in the `Listens(receiver, sender)` log record each pair of users where one receives messages sent by the other. For example, the entry (Alice, Bob) means that Alice receives all messages sent by Bob. Each receiver may listen to messages sent by many senders, and each sender may have many receivers. Users may receive their own messages, i.e., the entry (Pat, Pat) means that all messages sent by Pat are also received by Pat.

The other log, `Message(sender, contents)`, contains each message sent on the system and the name of the sender.

To simplify the problem you can assume that the Listens log is a complete list of the (receiver, sender) pairs during the entire time that all the messages were sent, and that its contents did not change. You may also assume there are no duplicate entries in either log.

Describe a sequence of one or more map-reduce jobs (not Pig programs) to count the number of messages received by each user. The output of the final job should contain one entry for each user and the number of messages received by that user, i.e., if (Chris, 4217) appears in the final output, it means that 4217 messages were received by Chris.

You need to clearly describe the (key, value) pairs that are input to and output from each map and reduce phase of the map-reduce job(s) needed. But the exact format is up to you – it can be pseudo-code or pseudo-java. But you need to clearly describe the input and output (key, value) pairs from each map and reduce stage, and explain how the output of each map or reduce stage is computed from its input. If it takes more than one map-reduce job to compute the final result you should show how the output of each job is used as input to subsequent ones.

This question turned out to be a little more complex than we anticipated because we need to do the map-reduce version of a join to solve it. It requires a series of map-reduce jobs, and the first jobs in the series write records containing "tagged" information that can be merged later.

M-R job 1. Compute the number of messages produced by each sender.

Map 1 input: Message log. Key = sender, Value = message (which we ignore)

Map 1 output: key = sender, value = 1

Reduce 1 input: key = sender, value = [1], i.e., array of 1 values representing messages sent

Reduce 1 output: key = sender, value = tuple ("count", size of sender's input array (= # messages sent)) [i.e., the output has, for each sender, the number of messages sent by that user plus the string "count" which acts as a tag.]

(more space is available on the next page for your answer if needed)

Question 8. (cont.) Additional space for your map-reduce algorithm if needed.

M-R job 2. Compute for each sender a list of the users receiving messages from that sender.

Map 2 input: Listens log. Key = receiver, value = sender

Map 2 output: key = sender, value = receiver

Reduce 2 input: key = sender, value = list of receivers

Reduce 2 output: key = sender, value = tuple ("receivers", list of receivers) [i.e., for each sender a tagged tuple with a list of all receivers.

M-R Job 3. Combine the information from the previous two jobs to find out how many messages are received by each listener.

Map 3 input: output files from jobs 1 and 2. Key = sender. Value is either ("count", number of messages sent), or ("receivers", list of receivers).

Map 3 output: For each receiver, key = receiver, value = number of messages sent to that receiver by a particular sender.

Reduce 3 input: key = receiver, value = array with counts of messages sent by different senders to that receiver.

Reduce 3 output: key = receiver, value = sum of counts of messages sent to that receiver.

Question 9. (8 points) The promised "what does this mean" question. ACID is a collection of four properties provided by database transactions. For each of the four, give a one- or two-sentence definition of what that term means. Please be very brief, but precise.

Atomic: **Transactions are "all or nothing" – either all parts of a transaction happen, or none of it does.**

Consistent: **Provided the application logic is correct, a transaction will leave the database in a consistent state, provided it started in one.**

Isolation: **Transactions execute as if they had occurred in some serial order isolated from each other.**

Durable: **Once a transaction has committed, the results of the transaction are permanent.**

Question 10. (12 points) Many of the no-SQL file systems that support large-scale parallel execution guarantee "eventual consistency" rather than the ACID properties.

(a) (6 points) What does this mean? Explain briefly how this can produce different results compared to a parallel SQL system that guarantees the ACID properties. A very short example might be helpful.

Eventual consistency guarantees that if there are multiple copies of a data object, and that object is updated, eventually all the copies will be changed to reflect the update. However that is not guaranteed to happen at once and old values of the data may be visible in some copies for some time after the change. If the data is read soon after the update, either the new or old value might be returned. That contrasts with a SQL system with isolation level serializable (i.e., ACID). In that situation, once a change has been committed, all new transactions will read only the new value.

(b) (6 points) Why is this done? Give a technical reason why no-SQL systems find it useful to provide eventual consistency rather than ACID guarantees.

Eventual consistency has less overhead than ACID. It is not necessary to lock and update all copies of a data item simultaneously when it is changed.