CSE 132A: Database System Principles
Summer Session I, 2013

Final (A)

(Friday, August 2)
Total: 28 points

Problem 1 *(8 points).* For each of the following statements **circle a single** correct answer (no justification required):

(a) In SQL, the query "SELECT name FROM person" over the relation person(name, address) **always** returns a set of tuples (i.e., it does not return duplicates).

TRUE **FALSE**

Explanation: It can return duplicates if name is not a key of relation person.

(b) In SQL, the following query can be flattened.
SELECT *
FROM driver, (SELECT * FROM rental WHERE VIN = 'ULTM')
WHERE driver.age >50

TRUE FALSE

Explanation: SQL queries with subqueries in the FROM clause only can always be flattened.

(c) In a relational DBMS, when a SQL query mentions a **materialized** view, the DBMS needs to do view unfolding to answer this query.

TRUE **FALSE**

Explanation: View unfolding is used to answer queries involving *virtual* views.

(d) The following two SQL queries Q_1 and Q_2 over the relation person(name, address) always return the same result:
Q_1: SELECT COUNT(*) FROM person
Q_2: SELECT COUNT(name) FROM person

TRUE **FALSE**

Explanation: They will return a different result if person contains tuples with null values for name, since the first query will count those tuples, while the second will not.

(e) Recursion increases the expressive power of SQL.

TRUE FALSE

Explanation: Without recursion SQL cannot express properties such as the transitive closure of a graph.

(f) The following two relational algebra expressions Q_1 and Q_2 over the relations R(x, y) and S(x, z) are equivalent (i.e., they return the same result on all possible instances).
Q_1: $\pi_{yz}(\sigma_{x1=x2}(\delta_{x\to x1}(R) \times \delta_{x\to x2}(S)))$
Q_2: $\pi_{yz}(R \bowtie S)$

TRUE FALSE

(g) The following two relational algebra expressions Q_1 and Q_2 over the relations R(x, y) and S(z, w) are equivalent (i.e., they return the same result on all possible instances). Note that R and S have different schemas from the previous question.

Q_1: $R \bowtie S$
Q_2: $R \times S$

TRUE FALSE

Explanation: If the schemas of R and S had any columns in common, these two relational algebra expression would *not* have been equivalent. However, in this case the schemas have no columns in common and therefore their natural join is equivalent to their cartesian product.

(h) Database Management Systems (DBMSs) contain a component that makes sure that a collection of operations is performed **atomically** as a **single** logical function (i.e., either all operations in the collection are performed or none of them). Select from the options below the component of the DBMS whose primary responsibility is to provide this functionality.

{Data Model} {Query Processor} **{Transaction Manager}** {Storage Manager}

(i) A relational query processor, upon receiving a SQL query, creates physical query plans (in short pqp), logical query plans (in short lqp) and a parse tree (in short pt). Pick from the options below the one that corresponds to the correct order in which these objects are created. An option {a, b, c} denotes that the query processor creates first a, then b and then c.

{pt, lqp, pqp} {pt, pqp, lqp} {lqp, pqp, pt} {pqp, lqp, pt}

Explanation: Upon receiving a SQL query, a relational query processor first creates a parse tree. Subsequently this is translated to a logical plan, which is rewritten to multiple logical plans. These logical plans are then translated to physical plans.

(j) Consider the following instance:

R	x	y
	1	b
	1	c
	2	c
	3	b

S	x	z
	1	b
	1	c
	2	b

and the following four SQL queries Q_1, Q_2, Q_3, and Q_4:
Q_1: SELECT * FROM R NATURAL INNER JOIN S

Q_2: SELECT * FROM R NATURAL LEFT OUTER JOIN S
Q_3: SELECT * FROM R NATURAL RIGHT OUTER JOIN S
Q_4: SELECT * FROM R NATURAL FULL OUTER JOIN S

Select from the options below the set of queries that return the **same** answer **on this particular instance**:

$\{Q_1, Q_2\}$ **$\{Q_1, Q_3\}$** $\{Q_1, Q_4\}$ $\{Q_2, Q_3\}$

Explanation: Consider an outer join between two relations. If for each of the tuples in the relation on the side of the outer, we can find a matching tuple (i.e., a tuple with the same value on the join attribute) in the other relation, then this outer join returns the same result as the inner join. In this case this is happening for the right outer join.

(k) Consider relations R(x, y) and S(x, z) and a condition p involving some of the attributes x, y, and z. Consider also the following rule:

$\sigma_p(R \bowtie S) \leftrightarrow (\sigma_p(R)) \bowtie S$

Select from the options below the set of attributes that can be involved in condition p so that the above rule is a valid rewrite rule.

{y only} **{x and y}** {z only} {x and z}

Explanation: To push the selection below the join onto the side of R, predicate p should involve only attributes of R (i.e., attributes x and y).

(l) Consider a relational schema with relation R(x, y) and an index on attribute y. Consider also the SQL query Q_1: "SELECT x FROM R WHERE y = 3". The following is a valid physical query plan for Q_1. (Remember that in class we have been writing query plans mostly as trees. However, they can also be written horizontally as expressions as shown below).

$\pi_x^{FLY}(\sigma_{y=3}^{INDEX}(R^{SCAN}))$

TRUE FALSE

(m) A **nested-loop join** between two relations runs faster when both relations are sorted.

TRUE **FALSE**

Explanation: Nested-loop join iterates over the entire relations without exploiting the fact that two relations are sorted. It is merge join (also know as sort-merge join) that exploits sorted input.

(n) Consider relations R(x, y), S(y, z), T(z, w), U(w, k) and the following relational algebra expression:

$\sigma_{x=2}(R) \bowtie S \bowtie T \bowtie U$

where $\sigma_{x=2}(R)$ creates a small relation. Select from the following options the order in which the joins will be executed according to the Wong-Youssefi algorithm (we assume that all assumptions of the Wong-Youssefi algorithm, including the existence of indexes on all join attributes hold).

$\boldsymbol{((\sigma_{x=2}(R) \bowtie S) \bowtie T) \bowtie U}$ $(\sigma_{x=2}(R) \bowtie S) \bowtie (T \bowtie U)$ $\sigma_{x=2}(R) \bowtie (S \bowtie (T \bowtie U))$

(o) Consider a relation R, x a subset of the attributes of R and p a condition involving some attributes of R. The following is always a valid rewrite rule:

$\pi_x(\sigma_p(R)) \leftrightarrow \sigma_p(\pi_x(R))$

TRUE **FALSE**

Explanation: This is not a valid rewrite rule, since on the left expression p may refer to attributes of R that are not among the set x of attributes.

(p) Consider the relation R(x) and the following relational calculus expression over this relation:

ϕ: $\{t : x \mid \forall r \in R\ (r(x) = t(x) \wedge r(x) > 1)\}$

Consider also the following database instance I:

R	x
	1
	2

Pick from the options below the set that corresponds to the answer of expression ϕ when evaluated over I. (In the notation below $\{<1>, <2>\}$ denotes the set of two tuples; the first having a single value '1' and the second having a single value '2'. Moreover, $\{\}$ denotes the empty set.)

{<1>, <2>} {<1>} {<2>} **{}**

Explanation: You have to pay attention to the universal quantifier: Due to its existence the above expression is returning any t tuple whose x value is equal to the x value of *every* tuple in R. Since there are two tuples in R with different x values, the returned t tuple should have an x value that is equal to two different values, which is obviously not possible. Therefore there is no such t tuple and the above expression returns the empty result when evaluated on I.

Problem 2 *(1 point).* In SQL, the keyword NOT IN can be simulated by the keyword NOT EXISTS. The goal of this exercise is to show how this can be done. Consider the following SQL query:

Q_1: SELECT x
FROM R
WHERE y NOT IN Q

where Q is some subquery in SQL that has (z) as its output schema. Rewrite Q_1 using NOT EXISTS. Obviously you should not be using IN or NOT IN in your solution.

Solution:

Checking that a value y is not among the z values returned by Q is equivalent to checking that there does not exist a z value returned by Q that is equal to y. Therefore the above query can be equivalently written as follows:

```
SELECT x
FROM R r
WHERE NOT EXISTS (
      SELECT *
      FROM Q q1
      WHERE q1.z = r.y)
```

Note the need to look for the existence of y values among the z values output by Q. Leaving out this condition simply checks for emptiness of Q. For instance the query "SELECT x FROM R WHERE NOT EXISTS (SELECT y FROM Q)" returns the x value of every tuple in R if and only if Q does not return a result, which is different from returning all x values from R tuples whose y values are among those output by Q (which is what the original query is computing).

Problem 3 *(9 points.)* Consider the database of a bank with the following schema:

customer(<u>name</u>, age)
account(<u>accountID</u>, amount, state)
hasaccount(<u>name, accountID</u>)

The relation *customer* lists the name and age of customers, the relation *account* lists the ID of an account, the amount of money deposited in the account and the state in which the account was opened, and finally the relation *hasaccount* contains tuples of the form (n, a) denoting that the customer with name = c has the account with accountID = a. The underlined attributes form the primary key of the corresponding relation. In the case of *hasaccount* all its attributes form its primary key. Additionally, *hasaccount.name* is a foreign key referencing *customer.name* and *hasaccount.accountID* is a foreign key referencing account.accountID. Below is an example instance over this schema. (Note that the relation names are abbreviated due to lack of space: *cust* stands for *customer*, acc stands for *account* and *hasacc* stands for *hasaccount*.)

Database instance:

cust	*name*	*age*
	Anna	18
	Tom	20
	Mary	30

acc	*accountID*	*amount*	*state*
	A1	800	CA
	T1	900	CA
	T2	2000	NY

hasacc	*name*	*accountID*
	Anna	A1
	Tom	T1
	Tom	T2

For each of the following expressions write the result of evaluating each expression on the above instance. Your answer **should not** be a description in english. It should be the table that is produced by the evaluation of the corresponding expression. For your help, the above database instance is repeated also on the top of the following page. *For your answers use the space below each question or on the back of the page.*

(a) $\pi_{name}(customer) - \pi_{name}(customer \bowtie hasaccount \bowtie account)$

Solution:

	name
	Mary

(b) $\pi_{name,accountID}(\sigma_{age<19}(customer) \times account)$

Solution:

	name	***accountID***
	Anna	**A1**
	Anna	**T1**
	Anna	**T2**

Database instance:

cust	*name*	*age*
	Anna	18
	Tom	20
	Mary	30

acc	*accountID*	*amount*	*state*
	A1	800	CA
	T1	900	CA
	T2	2000	NY

hasacc	*name*	*accountID*
	Anna	A1
	Tom	T1
	Tom	T2

(c) SELECT name
FROM customer NATURAL INNER JOIN hasaccount NATURAL INNER JOIN account
WHERE amount <1000

Solution:

	name
	Anna
	Tom

(d) SELECT name
FROM customer
WHERE name NOT IN
(SELECT name
FROM customer NATURAL INNER JOIN hasaccount NATURAL INNER JOIN account
WHERE amount >1000)

Solution:

	name
	Anna
	Mary

(e) $\{t : \text{name, accountID} \mid \exists c \in \text{customer}\ \exists h \in \text{hasaccount}\ \exists a \in \text{account}\ [$
$t(\text{name}) = c(\text{name}) \ \wedge\ t(\text{accountID}) = a(\text{accountID}) \wedge$
$c(\text{name}) = h(\text{name}) \ \wedge\ h(\text{accountID}) = \text{a}(\text{accountID}) \ \wedge\ c(\text{age}) \ <\ 21]\}$

Solution:

	name	***accountID***
	Anna	**A1**
	Tom	**T1**
	Tom	**T2**

(f) $\{t : \text{accountID} \mid \exists a1 \in \text{account}\ [t(\text{accountID}) = a1(\text{accountID}) \wedge$
$\forall a2 \in \text{account}\ [a2(\text{state}) = a1(\text{state}) \rightarrow a1(\text{amount}) \geq a2(\text{amount})]]\}$

Solution:

	accountID
	T1
	T2

Problem 4 *(6 points.)* Using the schema of problem 3 write the following queries. The queries should work **on any possible database instance over this schema**; not only on the example instance shown in problem 3. *For your answers use the space below each question or on the back of the page.*

(a) Write a query that computes the customers (identified by their name) whose **every** account (if any) is in CA. In other words, you should return all customers who either do not have any accounts or if they do have an account, they do not have any account outside CA. The output schema of the query should be (name). Write this query in each of the following languages:

(i) In SQL

Solution:

We need to select all customers who are not among those that have at least one account outside CA. This can be accomplished through the following query:

```
SELECT name
FROM customer
WHERE name NOT IN
      (SELECT name
      FROM hasaccount INNER JOIN account
      WHERE state <>'CA')
```

Please note the following:

(a) The above query is *not* equivalent to the query "SELECT name FROM hasaccount INNER JOIN account WHERE account <>'CA"', which returns customers that have at least one account in CA, but might also have other accounts outside CA.

(b) The problem is asking for customers who have all *their* accounts in CA. This is different from customers who own all CA accounts in the bank.

(c) To be entirely correct the query would need to also have an additional condition in the WHER clause checking that somebody does not have an account with a null value for state. However, since in relational calculus and algebra that are used later on we did not consider NULL values, the SQL query above was considered correct.

(ii) In relational algebra **or** relational calculus. You can select the one you are most comfortable with. Please provide **only one** of them (not both).

Solution:

Although on the exam you were asked to write either the relational algebra or relational calculus expression, below you can find both expressions:

(a) In relational algebra

We follow a similar reasoning as the one used for the SQL query above. We simply return all customers who not among those that own a non-CA account.

$\pi_{name}(customer) - \pi_{name}(hasaccount \bowtie \sigma_{state<>'CA'}(account))$

(b) In relational calculus

We should return names of customers such that everyone of their accounts is in CA. This can be expressed in relational calculus as follows:

$\{t : \textbf{name} \mid \exists c \in \textbf{customer}\ [t(\textbf{name}) = c(\textbf{name}) \wedge$
$\forall h \in \textbf{hasaccount}\ \forall a \in \textbf{account}$
$[(h(\textbf{name}) = c(\textbf{name}) \wedge h(\textbf{accountID}) = a(\textbf{accountID})) \rightarrow$
$a(\textbf{state}) = \textbf{'CA'}]]\}$

(b) Write a query that finds pairs of customers (customer1, customer2) identified by their name that do not have any account in the same state (i.e., customer1 does not have any account in the states where customer2 has an account and vice versa). In contrast to your project problem you should return a customer pair in both orders (i.e., if you return (c1, c2), you should also return (c2, c1)) but you should not return a pair of a customer with himself/herself (i.e., you should not return (c, c)). The output schema of the query should be (name1, name2). Write this query **in relational calculus**.

Solution:

We need to return pairs of names of different customers, such that for any pair of accounts where the first account is owned by the first customer and the second account is owned by the second customer, these accounts are in different states. This can be expressed in relational calculus as follows:

$\{t : \textbf{name1, name2} \mid \exists c1 \in \textbf{customer}\ \exists c2 \in \textbf{customer}$
$[t(\textbf{name1}) = c1(\textbf{name}) \wedge t(\textbf{name2}) = c2(\textbf{name}) \wedge c1(\textbf{name}) <> c2(\textbf{name}) \wedge$
$\forall h1 \in \textbf{hasaccount}\ \forall a1 \in \textbf{account}\ \forall h2 \in \textbf{hasaccount}\ \forall a2 \in \textbf{account}$
$[(h1(\textbf{name}) = c1(\textbf{name}) \wedge h1(\textbf{accountID}) = a1(\textbf{accountID}) \wedge$
$h2(\textbf{name}) = c2(\textbf{name}) \wedge h2(\textbf{accountID}) = a2(\textbf{accountID})) \rightarrow$
$a1(\textbf{state}) <> a2(\textbf{state})]]\}$

Problem 5 *(2 points.)* Consider the database schema of problem 3 and the following SQL query:

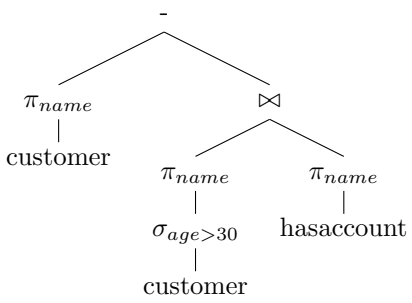
Q_1 : SELECT name
FROM customer
WHERE name NOT IN
(SELECT name
FROM customer, hasaccount, account
WHERE customer.name = hasaccount.name AND hasaccount.accountID = account.accountID
AND age >30)

Write a **logical** query plan for Q_1 that satisfies the following requirements:
(a) It does not contain any cartesian product operator.
(b) It has projections and selections pushed down as much as possible (i.e., it projects out attributes and applies selections as early as possible).

A logical plan that is a correct logical plan for Q_1 but does not satisfy either or both of the above requirements, will still receive partial credit.

Solution:

The query is asking for all customers that are not among the customers that own at least one account and are over the age of 30. A logical query plan for this query which does not employ cartesian products and has all projections and selections pushed down is shown below:



Note that the above query does not employ the account relation as its presence is implied by the foreign key from the hasaccount relation. During grading plans with the account relation or any other equivalent plans were considered correct as long as they satisfied both conditions (a) and (b) (which for example in the presence of the relation account would mean that immediately over the scan of the account relation the attribute amount should have been projected out).

Problem 6 *(2 points.)* Consider the database of a travel agency containing a relation flight that stores the **direct** flights that exist between two cities:

flight(departure: string, destination: string)

In this relation a tuple ('San Diego', 'Los Angeles') denotes the fact that there is a direct flight from San Diego to Los Angeles.

Using a recursive SQL view, write a SQL query that returns all pairs of cities (city1, city2), such that there is a way to fly from city1 to city2 with **at most** 100 stops. Obviously when there is a direct flight from city1 to city2, we say that there is a way to fly from city1 to city2 with 0 stops. The output schema of the query should be (departure, destination). Note that **you are expected to use a recursive view**; you **should not** write anything along the lines of a union of 100 queries! Moreover, note that it should be done using a **recursive view** and **not** by using embedded SQL. You are free to create an intermediate (recursive) view and then write another query that uses this view to compute the final result.

Solution:

The query is very similar to the recursive query we have seen in class. The only tricky part was enforcing the constraint on the length of the paths. This can be done by first creating a recursive view as follows:

```
CREATE RECURSIVE VIEW reachable as
(SELECT departure, destination, 0 AS stops
FROM flight)
UNION
(SELECT f.departure, r.destination, stops + 1 AS stops
FROM flight f, reachable r
WHERE f.destination = r.departure AND stops <100)
```

Then we can use this view to compute the final result using a simple query:

```
SELECT departure, destination
FROM reachable
```