CSE 344 Midterm

Monday, Nov 4, 2013, 9:30-10:20

Name: __________________________

Question	Points	Score
1	30	
2	10	
3	50	
4	10	
Total:	100	

- This exam is open book and open notes but NO laptops or other portable devices.
- You have 50 minutes; budget time carefully.
- Please read all questions carefully before answering them.
- Some questions are easier, others harder. Plan to answer all questions, do not get stuck on one question. If you have no idea how to answer a question, write your thoughts about the question for partial credit.
- Good luck!

1 SQL

1. (30 points)

   A *sparse* matrix is a matrix $A = (a_{ij})$ where many elements a_{ij} are 0. A sparse matrix can be stored in a relation with three attributes: $A(i, j, v)$ where i, j are the row and column and v the value of the element in that row and column. For example the matrix:

$$A = \begin{pmatrix} 30 & 0 & 0 \\ -10 & 0 & 0 \\ 0 & 10 & 50 \end{pmatrix}$$

   can be represented as:

i	j	v
1	1	30
2	1	-10
3	2	10
3	3	50

   Notice that it's OK to keep 0 values, even though it may be less efficient. For example, the matrix above can also be represented as:

i	j	v
1	1	30
1	2	0
1	3	0
2	1	-10
3	2	10
3	3	50

Reference for SQL Syntax

Outer Joins

```
-- left outer join with two selections:
select *
from R left outer join S on R.x=55 and R.y=S.z and S.u=99
```

The UNION Operation:

```
select R.k from R union select S.k from S
```

The CASE Statement:

```
select R.name, (case when R.rating=1 then 'like it'
                     when R.rating=0 then 'do not like it'
                     when R.rating is null then 'do not know'
                     else 'unknown' end)
               as my_rating
from R;
```

The WITH Statement

Note: `with` is not supported in sqlite, but it is supported SQL Server and in postgres.

```
with T as (select * from R where R.K>10)
  select * from T where T.K<20
```

Reference for the Relational Algebra

Cheat sheet for relational algebra

Name	Symbol
Selection	σ
Projection	π
Join	$\bowtie$
Group By	γ
Set Difference	$-$
Duplicate Elimination	δ

(a) (15 points) You are given two sparse matrices A, B with schemas:

```
A(i,j,v)
B(j,k,v)
```

Write an SQL query that computes the product matrix $A \cdot B$. Your query should return a set of triples (i, k, v) representing the product matrix; you do not need to remove the entries with value 0.

Recall that the product of an $m \times n$ matrix $A = (a_{ij})$ with an $n \times p$ matrix $B = (b_{jk})$ is the $m \times p$ matrix $C = (c_{ik})$ where $c_{ik} = \sum_{j=1,n} a_{ij} b_{jk}$. For example:

$$\begin{pmatrix} 30 & 0 & 0 \\ -10 & 0 & 0 \\ 0 & 10 & 50 \end{pmatrix} \cdot \begin{pmatrix} 0 & 0 & 0 \\ 0 & 2 & 4 \\ 0 & 0 & -1 \end{pmatrix} = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 20 & -10 \end{pmatrix}$$

Turn in a SQL query:

Solution:

```
select A.i,B.k, sum(A.v*B.v)
from A,B
where A.j=B.j
group by A.i,B.k
```

(b) (15 points) You are given two sparse matrices A, B with schemas:

`A(i,j,v)`
`B(i,j,v)`

Write an SQL query that, given two sparse matrices A, B computes the sum matrix $A+B$. Your query should return a set of triples (i, j, v) representing the sum matrix; you do not need to remove the entries with value 0.

For example:

$$\begin{pmatrix} 30 & 0 & 0 \\ -10 & 0 & 0 \\ 0 & 10 & 50 \end{pmatrix} + \begin{pmatrix} 0 & 0 & 0 \\ 0 & 2 & 4 \\ 0 & 0 & -1 \end{pmatrix} = \begin{pmatrix} 30 & 0 & 0 \\ -10 & 2 & 4 \\ 0 & 10 & 49 \end{pmatrix}$$

Turn in a SQL query:

Solution:

```
with T as (select i,j from A union select i,j from B)
select T.i, T.j, (case when A.v is null then 0 else A.v end) +
                 (case when B.v is null then 0 else B.v end)
from T left outer join A on (T.i = A.i and T.j = A.j)
       left outer join B on (T.i = B.i and T.j = B.j)
```

An elegant solution, from a student:

```
select x.i, x.j, sum(x.v)
from (select * from A union select * from B) as x
group by x.i, x.j
```

Note that postgres doesn't accept `A union B`, instead we had to say `select * from A union ...`; we did not take points off for `A union B`.

A full outer join does not work here. This is *incorrect*:

```
select A.i, A.j, (case when A.v is null then 0 else A.v end) +
                 (case when B.v is null then 0 else B.v end)
from A full outer join B on A.i = B.i and A.j = B.j;
```

It doesn't work because, for the entries `(B.i,B.j)` that do not have a matching entry in `A`, the values of `A.i,A.j` are NULL.

2 Indexes and Query Evaluation

2. (10 points)

   Consider the following two relations:

```
Person(pid, name)
Order(oid, pid, product, quantity, date)
```

   We create the following indexes:

```
create index Person_pid on Person(pid)
create index Person_name on Person(name)
create index Order_oid on Order(oid)
create index Order_pid on Order(pid)
create index Order_product on Order(product)
```

   Consider the following two SQL queries:

```
-- Q1
select product, quantity, date
from Person, Order
where Person.name = 'Alice' and Person.pid = Order.pid

-- Q2
select name
from Person, Order
where Person.pid = Order.pid and Order.product = 'Tablet'
```

   (a) (5 points) Which indexes may be useful for query Q1?

	Person_pid	Person_name	Order_oid	Order_pid	Order_product
Answer Y/N:					

Solution:

	Person_pid	Person_name	Order_oid	Order_pid	Order_product
Answer Y/N:	N	Y	N	Y	N

(b) (5 points) Which indexes may be useful for query `Q2`?

	`Person_pid`	`Person_name`	`Order_oid`	`Order_pid`	`Order_product`
Answer Y/N:					

Solution:

	`Person_pid`	`Person_name`	`Order_oid`	`Order_pid`	`Order_product`
Answer Y/N:	Y	N	N	N	Y

3 Datalog, Relational Calculus, Relational Algebra

3. (50 points)

Consider the following two relations:

```
Person(id, name)
Trusts(id1, id2)
```

(a) (10 points) Write a program in non-recursive datalog-with-negation that returns the id's and names of all persons who don't trust Alice. Turn in a datalog program:

Solution: Solution 1:

```
Ans(x,y) :- Person(x,y), Person(z,'Alice'), x!=z
```

Solution2:

```
NonAns(x) :- Trusts(x,z), Person(z,'Alice')
Ans(x,y) :- Person(x,y), not NonAns(x)
```

For this question and the next, 2 Points where taken off for adding `Person(x,y)` to the `NonAns` rule. 3 points where taken off for writing `Trusts(x,'Alice')`.

(b) (10 points) Write a program in non-recursive datalog-with-negation that returns the id's and names of all persons who trust only Alice. (In particular, your query should return all persons who don't trust anyone, e.g. persons who do not appear in `Trust`.) Turn in a datalog program:

Solution: Solution 1:

```
NonAnsw(x) :- Trusts(x,y), not Person(y,'Alice')
Ans(x,y) :- Person(x,y), not NonAns(x)
```

Solution 2:

```
NonAnsw(x) :- Trusts(x,y), Person(z,'Alice'), y != z
Ans(x,y) :- Person(x,y), not NonAns(x)
```

Solution 3:

```
NonAnsw(x) :- Trusts(x,y), Person(y,n), n != 'Alice'
Ans(x,y) :- Person(x,y), not NonAns(x)
```

(c) (20 points) Consider the following query in the relational calculus:

$$A(x,n) = \texttt{Person}(x,n) \land \forall y.(\texttt{Trusts}(x,y) \Rightarrow \forall z.(\texttt{Trusts}(y,z) \Rightarrow \neg\texttt{Person}(z,\texttt{'Alice'})))$$

i. Write this query in non-recursive datalog-with-negation. Turn in a datalog program:

Solution:

```
    U(x) :- Trusts(x,y), Trusts(y,z), Person(z,'Alice')
    A(x,n) :- Person(x,n), not U(x)
```

Many people wrote two rules for `U` and that's OK:

```
   V(y) :- Trusts(y,z), Person(z,'Alice')
   U(x) :- Trusts(x,y), V(y)
```

We took 2 points off for each misplaced `not` (this easily added to 4 points); 1 point off for extra `Person(...)` predicates.

ii. Write this query in SQL. Turn in a SQL query:

Solution:

```
select p.id, p.name
from Person p
where not exists
           (select *
            from Trusts t1, Trusts t2, Person q
            where p.id = t1.id1 and t1.id2 = t2.id1
              and t2.id2 = q.id and q.name = 'Alice')
```

In many cases the datalog program was incorrect but the SQL query correctly implemented the datalog program; in that case I gave full credit.

(d) (10 points) Consider the following SQL query:

```
select distinct t1.id1
from Trusts t1, Person p1
where t1.id2 = p1.id and p1.name = 'Alice' and
  not exists (select *
              from Trusts t2, Person p2
              where p1.id = t2.id1 and t2.id2 = p2.id and p2.name = 'Bob')
```

Write this query in the Relational Algebra. Turn in a Relational Algebra plan:

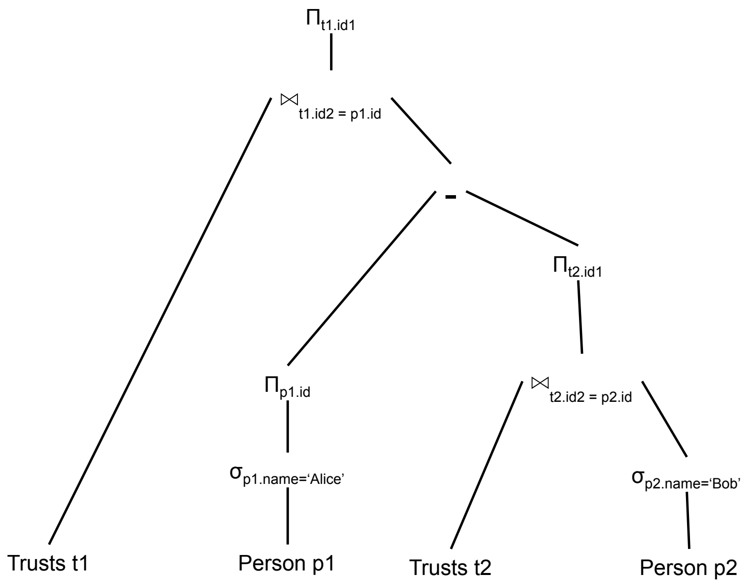
Solution: Write it first in datalog (it's much easier to read this way):

```
V(y) :- Trusts(y,z), Person(z,'Bob')
Q(x) :- Trusts(x,y), Person(y,'Alice'), not V(y)
```

Next, expose the difference clearly:

```
V(y) :- Trusts(y,z), Person(z,'Bob')
U(y) :- Person(y,'Alice'), not V(y) -- here's the set difference
Q(x) :- Trusts(x,y), U(y)
```

The query plan is:



4 XML and XPath

4. (10 points)

For each statement below indicate whether it is true or false.

(a) (1 point) XML is better suited than the relational data model for data exchange between applications.

(a) **Yes**

(b) (1 point) The data model for XML is a tree.

(b) **Yes**

(c) (1 point) B^+ trees are stored in XML

(c) **No**

(d) (1 point) An XML tree can be at most 6 levels deep.

(d) **No**

(e) (1 point) The tuples in a relation are unordered.

(e) **Yes**

(f) (1 point) The elements in an XML document are unordered

(f) **No**

(g) (1 point) The *Data Independence* principle states that the tuples in a relation must be independent of each other.

(g) **No**

(h) (1 point) The *Data Independence* principle states that applications should be written independently of the way the data is stored and organized.

(h) **Yes**

(i) (1 point) All elements returned by the XPath expression `/a//b[c=4]/d` are of type `<d>...</d>`.

(i) **Yes**

(j) (1 point) If the XPath expression `/a//b[c=4]` returns 10 elements, then the XPath expression `/a//b/c` must return at least 10 elements.

(j) **Yes**