Problems

Sunday, April 28, 2019 9:55 PM

```
Employee(eid, name, salary)
Project(pid, title, budget)
WorksOn(eid, pid, year)
```

2017 Aut ★★★★

(b) (10 points) We say that an employee worked intermittently on a project p if she worked on p during one year, then did not work on p during a later year, then worked again on p during an even later year. For example if Alice worked on the project during 2012, 2013, and 2017 then we say that she worked intermittently on p; if Bob worked on that project during 2013, 2014, 2015 and no other years, then we say he worked continuously. Write a SQL query to retrieve all employees that worked intermittently on some project. Return the employee name, and the project title.

Solution:

```
select distinct u.name, v.title
from Employee u, WorksOn x, WorksOn y, Project v
where u.eid = x.eid and u.eid = y.eid
  and x.pid = v.pid and y.pid = v.pid
  and x.year + 1 < y.year
  and not exists (select *
                  from WorksOn z
                  where u.eid = z.eid and v.pid = z.pid
                    and x.year < z.year and z.year < y.year);
```

(1 point) Distinct output generated
(3 Points) Basic query structure
- Selected the name and title
- FROM included all tables
- Proper joins created between tables

(1 point) Included check for year separation (ex: $year1 > year2 + 1$)
(5 points) Verification of intermittency (ex: subquery to check for years between separated years)

There were also variations on the solutions that could have worked (or did work). We took points off as similar to the original rubric as we could.

★★★ 2017 Aut

ii. (2 points) Are Q3 and Q4 equivalent? Careful

```
Q3: select W.year, W.pid, count(*)
    from WorksOn W, Employee E
```

```
    where W.eid = E.eid and E.salary > 100000
    group by W.year, W.pid;

Q4: select W.year, W.pid,
           (select count(*)
            from Employee E
            where W.eid = E.eid and E.salary > 100000)
    from WorksOn W;
```

ii. **No**

Yes/No:

★★★★★

2017 Aut

```
Employee(eid, name, salary)
Project(pid, title, budget)
WorksOn(eid, pid, year)
```

(b) (10 points) A project p1 influences a project p2, if there exists an employee who worked on p1 during some year, then worked on p2 during some later year. After a major bug was discovered in several projects, the company traced it down to a design flaw in the ''Compiler'' project, and now wants to retain only the projects that were not influenced by ''Compiler''. (Note ''Compiler'' *is* influenced by ''Compiler''.) Write a datalog program to find all projects who were not influenced by the ''Compiler'' project; return their pid and title.

Solution:

```
Infl(x) :- Project(x, 'Compiler', -)
Infl(z) :- Infl(x),
           WorksOn(eid, x, y1),
           WorksOn(eid, z, y2),
           y1 < y2
Answ(pid, title) :- Project(pid, title, -), !Infl(pid)
```

5 points off for missing recursion (almost nobody wrote a recursive query!)
2 points off for unsaafe
1 point off for missing $y1 < y2$
1 point off for missing negation

- 2017 Aut True/False

I. All queries expressible in RA are monotone. F.

If we can't write it as a SELECT-FROM-WHERE query without nested subqueries, then it's not monotone.

II. All queries expressible in datalog with ... are ... monotone.

II. Can queries expressible in Datalog (with recursion, but without negation and without aggregates) are monotone.

True.

III. NoSQL system support physical data independence better than RA systems.

False.

- 2019 Win ★★

The key-value pair data model is better suited for complex queries (with aggregates) than for simple queries (single lookups).

False.

- ★★ 15Win

The International Sled Dog (Husky) Racing Association (ISDRA) uses the following tables to store information about previous sled races[1]:

```
Dogs    (did integer, name string, age integer)
Mushers (mid integer, name string) -- mushers are drivers of sleds

Races   (mid integer, did integer, raceNumber int)
-- mid and did are foreign keys to Mushers and Dogs, respectively
```

Sample tuples from the tables:

be careful

```
Dogs:    (1, "Harry"), (2, "Dubs")
Mushers: (10, "Mary Shields"), (11, "Rick Swenson")
Races:   (10, 1, 1), (11, 2, 1)
```

Write a SQL statement that computes each of the following:

a) Show the SQL statements for creating the Races table (5 points).

```
CREATE TABLE Races (mid INTEGER REFERENCES Mushers,
                    did INTEGER REFERENCES Dogs,
                    raceNumber INTEGER,
                    PRIMARY KEY(mid, did, raceNumber));
```

d) Find the names of mushers who have raced with all the dogs in the past.

```
SELECT M.name
FROM   Mushers M
WHERE  NOT EXISTS (SELECT D.did
                   FROM   Dogs D
                   WHERE  NOT EXISTS (SELECT R.did
                                      FROM  Races R
                                      WHERE R.did = D.did AND R.mid = M.mid))
```

★★★★★ Are they equivalent?

$\pi_{R.c}((\sigma_{A>5}(R) \bowtie_{A=C} \sigma_{C<5}(S))$ | $(\sigma_{A>5}(R) \bowtie_{A=C} S) \bowtie_{B=E} T$

$\sigma_{B<G}((\sigma_{A<B}(R) \bowtie_{A=C} \sigma_{C>D}(S)) \bowtie_{D=F} \sigma_{F>G}(T))$

$(\sigma_{A<B}(R) \bowtie_{A=C} S) \bowtie_{D=F} T$

$(R \bowtie_{A=C} (S \bowtie_{D=F} T))$

Q<F=D<A=C<B

$\sigma_{B<Q} \rightarrow$ Empty

A=C<B $\bowtie_{D=F}$ $T \rightarrow T'(A=C, D=F)$

Empty

18ad ★★★★

```
Project(pid,name,startYear)
Developer(did,name,hireYear)
WorksOn(pid,did,year)
```

(e) (10 points) 'SystemX' is the oldest project of the company. Write a SQL query that returns all developers who worked every year on 'SystemX', from when it started until 2015. Your query should return the developers' ID and name.

the year project lasts

Solution:

```
-- Solution 1
SELECT Z.did, Z.name
FROM Project AS X, WorksOn AS Y, Developer AS Z
WHERE X.pid = Y.pid AND Y.did = Z.did
       AND X.startYear <= Y.year
       AND Y.year <= 2015
       AND X.name = 'SystemX'
GROUP BY Z.did, Z.name, X.startYear
HAVING count(*) = 2015 - X.startYear + 1;

-- Solution 2 (imperfect, but got full credit where present)
SELECT Z.did, Z.name
FROM Developer Z
WHERE NOT EXISTS (  -- find a year where Z didn't work on SystemX
  SELECT *
  FROM WorksOn Y    -- proxy for the set of all years (which we don't have
  WHERE NOT EXISTS (  -- check if Z worked that year on SystemX
     SELECT * FROM WorksOn Y2, Project X
     WHERE Y.year = Y2.year and Y2.pid = X.pid and X.name = 'SystemX'));
```

A solution without any negation got 2-3 points max.

A solution with only 1 negation got 5 points max.

1 point off for unnecessary join

★★ 18 au

(j) (1 point) This relational algebra expression is monotone: $\sigma_{\texttt{not(year=2015)}}(\texttt{WorksOn})$.

(j) **true**

True/False:

When you add tuples the output can never decrease

```
Project(pid,name,startYear)
Developer(did,name,hireYear)
WorksOn(pid,did,year)
```

(b) i. (2 points) Which of the following is the most accurate English interpretation of the SQL query below?

```
SELECT X.did
FROM Developer X
WHERE NOT EXISTS
          (SELECT *
           FROM Project Z
           WHERE NOT EXISTS
                   (SELECT *
                    FROM WorksOn Y
                    WHERE X.did = Y.did AND Y.pid = Z.pid
                      AND Y.year = 2015));
```

```
Developers that...
A: in 2015, didn't work on any projects at all
B: in 2015, didn't work on at least one of the projects
C: in 2015, worked on every single project
D: in 2015, worked on at least one project
```

i. **C**

A/B/C/D:

1. (50 points)

Consider the following database an online video rental service:

```
Customer(cid, name, country)
Movie(mid, title, year)
Rent(cid, mid, date)
Rate(cid, mid, score)
```

The `Customer` relation store all customers of the rental service, their names and the countries where they are located.

`Movie` stores all movies available for rent, their titles and the year of production.

When a customer rents a movie, a record is inserted in the relation `Rent`; a customer is allowed to rent a movie only once.

After watching the movie, a customer is allowed to rate the movie, with a score from 1 (worst) to 5 (best): this information is stored in the `Rate` table. Notice that a customer may rate a movie only if she/he has also rented the movie.

All primary keys are underlined. The attributes' types are as follows:

- `cid, mid, year, score` are integers.
- `name, country, title, date` are text.

```
Customer(cid, name, country)
Movie(mid, title, year)
Rent(cid, mid, date)
Rate(cid, mid, score)
```

(d) (10 points) Write a SQL query that returns the cid's and names of all customers who rented all the movies where "Alice" gave a score of 5. For example, if Alice gave a score of 5 to "Mad Max" and "Hunger Games" and gave no other score of 5, then your query should return the customers who rented both "Mad Max" and "Hunger Games".

Solution:

```
select *
from Customer x
where not exists
    (select *
     from Customer a, Rate b, Movie z
     where a.cid = b.cid and b.mid = z.mid
       and a.name = 'Alice' and b.score = 5
       and not exists (select * from Rent y
                       where x.cid = y.cid and y.mid = z.mid));
```

4 points off for one missing negation
8 points off for both missing negations
3 points off for wrong order of negated subqueries

```
QueryLog(uid,qid,word)
User(uid,name, language)
Dictionary(word, language)
```

(c) (15 points) You want to identify users who speak a foreign language. We assume that a user speaks a language L if she issues at least one query Q where all words are in L; if L is different from the user's native language (stored in `Users`) then we assume that she speaks the foreign language L. Write a query that returns all users that speak a foreign language. Your query should return the `uid`, the user name, and the foreign language.

For example, in our toy database your query should return `Jaque`, because he issued the query `day trip` where all words are in `English`; you should not return `Joe`: for example in his query `sidereal day`, while the first word is in Latin, the second is only in his native `English`.

```
select distinct x.uid, x.name, z.language
from User x, QueryLog y, Dictionary z  -- y defines the query Q
                                       -- z defines the language L
where x.uid = y.uid and y.word = z.word and x.language != z.language
  and not exists  -- check that ALL words in Q are in the language L
       (select *
        from QueryLog y1        -- is there a word NOT in L?
        where y.uid = y1.uid and y.qid = y1.qid
                                -- same user, same query
          and not exists
                    (select *
                     from Dictionary z1
                     where z1.word = y1.word
                       and z1.language = z.language));
```

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

(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.