

Midterm EXAMINATION

solutions

PROBLEM 1. (This problem uses the YCVB database, see last page of exam.)

Let ORDER-INFO be a relation as follows:

$$\pi_{\text{CUST, ITEM, QUANTITY}} (\text{ORDERS } \textit{join} \text{ INCLUDES})$$

(a) What does this query find? (i.e., write an equivalent query in English).

List all the items each customer ever ordered, in every possible quantity.

(b) How many tuples are in ORDER-INFO?

There are 6 pairs:

```
(Zack Zebra, Brie, 3)
(Zack Zebra, Perrier, 6)
(Zack Zebra, Macadamias, 2048)
(Ruth Rhino, Brie, 5)
(Ruth Rhino, Escargot, 12)
(Ruth Rhino, Endive, 1)
```

(q) Show an SQL query that computes ORDER-INFO.

```
SELECT o.CUST, i.ITEM, i.QUANTITY
FROM ORDERS o, INCLUDES i
WHERE o.ONUM = i.ONUM
```

PROBLEM 2. (This problem uses the YCVB database, see last page of exam.)

Consider the following query Q1:

$$\{c.name, o.item \mid \text{order-info}(o) \wedge \text{customers}(c) \wedge c.name = o.cust \wedge \\ \neg \exists o2 (\text{order-info}(o2) \wedge o2.item = o.item \wedge o2.quantity > o.quantity)\}$$

(a) What does Q1 find? (i.e., write an equivalent query in English).

This query finds the customers who've ordered the biggest quantity of some item, as well as the item they've ordered.

(b) Rewrite Q1 in domain relational calculus (DRC), without using '¬' or '≠' and without involving the CUSTOMERS relation.

$$\{name, item \mid \exists quantity (\text{order-info}(name, item, quantity) \wedge \forall quantity2, name2: \\ (\text{order-info}(name2, item, quantity2) \rightarrow (quantity2 \leq quantity))) \}$$

(c) Is Q1 safe? *Justify* your answer.

Safe TRC is defined on p. 157-158 of the Ullman Handout. Q1 satisfies all the criteria for a safe query:

1. There are no uses of the \forall quantifier or disjunction
2. Every free variable in a conjunction is limited (by one of the conjuncts).
3. Negation only applies to limited variables.

Therefore Q1 is safe.

PROBLEM 3.

We have a PEOPLE database, with a relation PERSON (with attributes *SSN*, *Name*, and *Gender*, in that order), and a relation PARENT (with attributes *ParentSSN* and *ChildSSN*, in that order)

- (a) Find a Relational Calculus query (either DRC or TRC) that finds all people with exactly one child.

```
{name: ∃ pssn, g (PERSON(pssn, name, g) ∧ PARENT(pssn, cssn) ^
                ¬∃ cssn2 (PARENT(ssn, cssn2) ^ cssn2 ≠ cssn)) }
```

- (b) Find the Datalog query that finds the names of all people whose children are all boys (i.e., they have at least one child, but no girls). Please try to choose meaningful variable / predicate names for your queries.

```
has_girls(Pssn) :- parent(Pssn, Cssn), person(Cssn, _, "female").
has_boys_only(Name) :- person(Pssn, Name, _), parent(Pssn, _), ~has_girls(Pssn).
```

PROBLEM 4.

We have a UConn database with a relation ONE_PREREQ containing info on pairs of courses where the second is a prerequisite of the first:

ONE_PREREQ relation:	COURSE1	COURSE2
	CSE263	CSE228
	CSE263	CSE245
	CSE124	CSE123
	CSE245	CSE221

We want to write a query ALL_PREREQS that returns a new relation listing, for each course, *all courses that need to be taken before it*:

ALL_PREREQS relation:	COURSE	PREREQS
	CSE263	CSE228
	CSE263	CSE245
	CSE263	CSE221
	CSE124	CSE123

- (a) Give 2 examples of similar queries that we've seen this semester.

- List, for each person, all their ancestors.
- List, for each part, all its subparts.

We decide to compute ALL_PREREQS from part (a) with a fixpoint operation:

```
ALL_PREREQS = X;
do
  ALL_PREREQS = Q /* code goes here */
until there is no change to ALL_PREREQS
```

where Q is a traditional relational query that uses the ALL_PREREQS relation as one of its inputs.

- (b) What is the initial value X of ALL_PREREQS? What is Q (write it in SQL)?

X is ONE_PREREQ; Q is as follows:

```

SELECT *
FROM ONE_PREREQ

UNION ALL

SELECT t1.COURSE, t2.PREREQS
FROM ONE_PREREQ as t1, X as t2
WHERE t1.PREREQS = t2.COURSE

```

- (c) Suppose we are told that no course ever has more than four prerequisites. Can we then express ALL_PREREQS without using fixpoint semantics? Explain why this information makes a difference. *Be specific.*

The reason that fixpoint semantics are needed in the general case is precisely because there is no bound on the length of the paths that we have to find (in this case, the paths of prerequisites). If we know that no course has more than four prerequisites, it gives us a bound of 4 on path length, and we can then express ALL_PREREQS in relational calculus as follows:

$$\begin{aligned}
 \text{ALL_PREREQS} = \{x, y \mid & \exists z_1, z_2, z_3 (\text{ONE_PREREQ}(x, y) \vee \\
 & (\text{ONE_PREREQ}(x, z_1) \wedge \text{ONE_PREREQ}(z_1, y)) \vee \\
 & (\text{ONE_PREREQ}(x, z_1) \wedge \text{ONE_PREREQ}(z_1, z_2) \wedge \text{ONE_PREREQ}(z_2, y)) \vee \\
 & (\text{ONE_PREREQ}(x, z_1) \wedge \text{ONE_PREREQ}(z_1, z_2) \wedge \text{ONE_PREREQ}(z_2, z_3) \wedge \text{ONE_PREREQ}(z_3, y)) \\
 &) \}
 \end{aligned}$$

PROBLEM 5.

Consider the program (where ~ represents negation)

```

c :- a, ~b.
a :- ~b.
b :- a, ~c.

```

- (a) What are all the fixpoints of this program?

There is only one fixpoint: a = TRUE, b = FALSE, c = TRUE. This is a fixpoint because, if we assign these values to the variables in the program and then evaluate the program, we obtain the same values for output:

```

c = a and not b = TRUE and not FALSE = TRUE
a = not b = not FALSE = TRUE
b = a and not c = TRUE and not TRUE = FALSE

```

It can be checked that the same is not true for any other assignment of values to these (boolean) variables.

- (b) Is there a least fixpoint? If yes, what is it? If no, why not?

A least fixpoint, if it exists, can be obtained by starting with an empty relation (when a,b,c are all FALSE) and iterating the query until we converge. In this case, if we do this, we indefinitely alternate between two solutions without converging. Therefore, there is no least fixpoint. Least fixpoints are not guaranteed to exist in programs that combine negation and recursion only when they are stratified, which is not the case here.

- (c) Is *aggregation* a monotonic operator or not? Prove your answer, based on definitions.

Aggregation involves two things: 1) partitioning the tuples into groups, and 2) applying the aggregation function to each group. The partitioning is monotonic: if we add tuples to the input relation, the size of each partition can only increase. Many aggregation functions are monotonic as well. For example, if we add tuples to the group, the value of MAX for the group can only go up. However, not all aggregation functions are monotonic; we can always define a new

one that is explicitly not monotonic. Therefore, the aggregation operator is not necessarily monotonic.