CPS 196.3 Fall 2003
Homework #2
Assigned: Wednesday, September 17
Due: Monday, September 29

Problem 1.

Impressed by your database designs for Zergs and Protoss, the Terran Space Commission asks you to revise an old, buggy design for a human genealogical database. List any problems with the relational design shown below, and then show your own relational design in SQL. Declare any reasonable constraints. State clearly any assumptions that you make about humans. (*Hint:* You only really need one table.)

```
CREATE TABLE Mother (id INTEGER NOT NULL PRIMARY KEY,
                     name VARCHAR(50) NOT NULL,
                     dob DATE NOT NULL,
                     childID INTEGER NOT NULL);
CREATE TABLE Father (id INTEGER NOT NULL PRIMARY KEY,
                     name VARCHAR(50) NOT NULL,
                     dob DATE NOT NULL,
                     childID INTEGER NOT NULL);
```

Problem 2.

Consider again the beer drinker's database from Homework #1. Key columns are underlined.

Drinker (*name*, *address*), *Bar* (*name*, *address*), *Beer* (*name*, *brewer*),
Frequents (*drinker*, *bar*, *times_a_week*), *Likes* (*drinker*, *beer*), *Serves* (*bar*, *beer*, *price*).

Run `/home/dbcourse/examples/db-beers/setup.sh` to setup a database with some sample data. For the SQL database schema, please refer to the file `create.sql` in the same directory. Write SQL statements to answer the following queries. *Make sure that result rows are* ***ordered*** *and there are* ***no duplicates****. Only use* `DISTINCT` ***when needed****; you may lose points if it is used unnecessarily.*

Write all your queries in a file named `hw2-2.sql`. When you are done, run "`db2 -tf hw2-2.sql > hw2-2.out`" (you may need to run "`db2 connect to cps116`" before that and "`db2 disconnect all`" afterwards). Then, print out files `hw2-2.sql` and `hw2-2.out` and turn them in together with the rest of the assignment.

(a) Find all drinkers who frequent James Joyce Pub.
(b) Find all bars that serve both Amstel and Corona.
(c) Find all bars that serve at least one of the beers Amy likes for no more than $2.50.
(d) For each bar, find all beers served at this bar that are liked by none of the drinkers who frequent that bar.
(e) Find all drinkers who frequent *only* those bars that serve some beers they like.
(f) Find all drinkers who frequent *every* bar that serves some beers they like.

(g) Find those drinkers who enjoy exactly the same set of beers as Amy.
(h) For each beer, find the bars that serve it at the lowest price.
(i) For each beer, find its average price and popularity (measured by the number of drinkers who like it). Sort the output by average price.
(j) Every time when Dan goes to a bar, he buys a bottle of the most expensive beer he likes that is served at this bar. If there is more than one such beer, he buys just one of them. If the bar does not serve any beer he likes, he will not buy any beer. Find the amount of money Dan spends every week buying beers in bars.

Problem 3.

Assume that in relational algebra, you can use built-in SQL predicates on strings, times, etc. in selection and join conditions; however, no SQL aggregation functions are allowed. Consider parts (g)-(j) of Problem 2:

(a) Which queries *cannot* be formulated in relational algebra?
(b) For each query that can be formulated in relational algebra, show the equivalent relational algebra query.

Problem 4.

Consider the following 21 query forms. *A*1 and *A*2 are columns; *Alist* is a comma-separated list of columns; *Rlist*, *Rlist*1, and *Rlist*2 are disjoint comma-separated lists of database tables; *Cond*, *Cond*1 and *Cond*2 are conditions that do not contain subqueries. You may assume that all attribute names are unique across the entire database. Note the *Cond*2 may include references to attributes from relations in *Rlist*1, i.e., references to relations outside the subquery.

1. Plain query:
```
SELECT DISTINCT Alist FROM Rlist WHERE Cond;
```
2. IN subquery:
```
SELECT DISTINCT Alist FROM Rlist
WHERE Cond1
AND A1 IN (SELECT DISTINCT A2 FROM Rlist2 WHERE Cond2);
```
3. NOT IN subquery:
```
SELECT DISTINCT Alist FROM Rlist
WHERE Cond1
AND A1 NOT IN (SELECT DISTINCT A2 FROM Rlist2 WHERE Cond2);
```
4. EXISTS subquery:
```
SELECT DISTINCT Alist FROM Rlist
WHERE Cond1
AND EXISTS (SELECT DISTINCT * FROM Rlist2 WHERE Cond2);
```
5. NOT EXISTS subquery:
```
SELECT DISTINCT Alist FROM Rlist
WHERE Cond1
AND NOT EXISTS (SELECT DISTINCT * FROM Rlist2 WHERE Cond2);
```

6. = ALL subquery:
```
SELECT DISTINCT Alist FROM Rlist
WHERE Cond1
AND A1 = ALL (SELECT DISTINCT A2 FROM Rlist2 WHERE Cond2);
```
7. NOT = ALL subquery:
```
SELECT DISTINCT Alist FROM Rlist
WHERE Cond1
AND NOT A1 = ALL (SELECT DISTINCT A2 FROM Rlist2 WHERE Cond2);
```
8. <> ALL subquery:
```
SELECT DISTINCT Alist FROM Rlist
WHERE Cond1
AND A1 <> ALL (SELECT DISTINCT A2 FROM Rlist2 WHERE Cond2);
```
9. NOT <> ALL subquery:
```
SELECT DISTINCT Alist FROM Rlist
WHERE Cond1
AND NOT A1 <> ALL (SELECT DISTINCT A2 FROM Rlist2 WHERE Cond2);
```
10. < ALL subquery:
```
SELECT DISTINCT Alist FROM Rlist
WHERE Cond1
AND A1 < ALL (SELECT DISTINCT A2 FROM Rlist2 WHERE Cond2);
```
11. NOT < ALL subquery:
```
SELECT DISTINCT Alist FROM Rlist
WHERE Cond1
AND NOT A1 < ALL (SELECT DISTINCT A2 FROM Rlist2 WHERE Cond2);
```
12. <= ALL subquery:
```
SELECT DISTINCT Alist FROM Rlist
WHERE Cond1
AND A1 <= ALL (SELECT DISTINCT A2 FROM Rlist2 WHERE Cond2);
```
13. NOT <= ALL subquery:
```
SELECT DISTINCT Alist FROM Rlist
WHERE Cond1
AND NOT A1 <= ALL (SELECT DISTINCT A2 FROM Rlist2 WHERE Cond2);
```
14. = ANY subquery:
```
SELECT DISTINCT Alist FROM Rlist
WHERE Cond1
AND A1 = ANY (SELECT DISTINCT A2 FROM Rlist2 WHERE Cond2);
```
15. NOT = ANY subquery:
```
SELECT DISTINCT Alist FROM Rlist
WHERE Cond1
AND NOT A1 = ANY (SELECT DISTINCT A2 FROM Rlist2 WHERE Cond2);
```
16. <> ANY subquery:
```
SELECT DISTINCT Alist FROM Rlist
WHERE Cond1
AND A1 <> ANY (SELECT DISTINCT A2 FROM Rlist2 WHERE Cond2);
```

17. `NOT <> ANY` subquery:
```
SELECT DISTINCT Alist FROM Rlist
WHERE Cond1
AND NOT A1 <> ANY (SELECT DISTINCT A2 FROM Rlist2 WHERE Cond2);
```
18. `< ANY` subquery:
```
SELECT DISTINCT Alist FROM Rlist
WHERE Cond1
AND A1 < ANY (SELECT DISTINCT A2 FROM Rlist2 WHERE Cond2);
```
19. `NOT < ANY` subquery:
```
SELECT DISTINCT Alist FROM Rlist
WHERE Cond1
AND NOT A1 < ANY (SELECT DISTINCT A2 FROM Rlist2 WHERE Cond2);
```
20. `<= ANY` subquery:
```
SELECT DISTINCT Alist FROM Rlist
WHERE Cond1
AND A1 <= ANY (SELECT DISTINCT A2 FROM Rlist2 WHERE Cond2);
```
21. `NOT <= ANY` subquery:
```
SELECT DISTINCT Alist FROM Rlist
WHERE Cond1
AND NOT A1 <= ANY (SELECT DISTINCT A2 FROM Rlist2 WHERE Cond2);
```

(a) For each query type except (1), if you can write an equivalent query using one of the other query types, show the equivalent query. You may assume that there are no `NULL` values in any of the relations.

(b) Give a minimal set of query types that is sufficient to express queries in all 21 forms. By a "minimal" set we mean that taking any type out of this set would produce a strict loss in expressive power, i.e., not all 21 forms would be expressible. You should include query type (1) in this minimal set.

Problem 5.

Below is the basic design for a used-car sales database. Key columns are underlined. Each automobile has a VIN (vehicle identification number), a model (e.g., Camero), a make (e.g., Chevrolet), a year (e.g., 1999), a color (e.g., red), a mileage (e.g., 50,000 miles), and a body style (e.g., coupe). Each automobile has a seller, which may be either a dealer or an individual. For each dealer, the database stores name, address, phone number. For each individual, only phone number and email address are recorded.

Automobile (*VIN*, *model*, *make*, *year*, *color*, *mileage*, *body_style*, *sellerID*)
Dealer (*sellerID*, *name*, *address*, *phone*)
IndividualSeller (*sellerID*, *phone*, *email*)

Keep all SQL statements you write for this problem in a file named `hw2-5.sql`. You can use "`@`" instead of "`;`" as the statement termination character in this case because of the triggers you are going to write in (b). When you are done, run "`db2 –td@ -f hw2-5.sql > hw2-`

5.out". Then, print out files hw2-5.sql and hw2-5.out and turn them in together with the rest of the assignment.

(a) Create the schema according to the given basic design, using CREATE TABLE statements. Choose appropriate data types for your columns, and remember to declare any keys, foreign keys, NOT NULL, and CHECK constraints when appropriate.

(b) Note that any *Automobile.sellerID* must be a *Dealer.sellerID* or *IndividualSeller.sellerID*. Also, a *Dealer.sellerID* cannot be an *IndividualSeller.sellerID*, and vice versa. It is not possible to declare these constraints as straightforward key and foreign key constraints. Instead, write triggers to reject any database modification that could violate these constraints.

Syntax for creating triggers in DB2 differs slightly from the standard SQL syntax presented in lecture. Please refer to http://www.cs.duke.edu/courses/fall03/cps196.3/faqs/sql.html for details.

(c) Start with empty tables. Write INSERT, UPDATE, and DELETE statements to illustrate that the triggers you wrote for (b) are working properly. More specifically:

- The first statement should attempt to insert a row into *Automobile* but should be rejected by your triggers.
- The second statement should insert a row into *Dealer* successfully.
- The third statement should attempt to insert a row into *IndividualSeller* but should be rejected by your triggers.
- The fourth statement should insert a row into *IndividualSeller* successfully.
- The fifth statement should insert a row into *Automobile* (with *sellerID* referring to a *Dealer*) successfully.
- The sixth statement should update the *Automobile* row's *sellerID* to refer to an *IndividualSeller* successfully.
- The seventh statement should attempt to update the *Automboile* row's *sellerID* but should be rejected.
- The eighth statement should attempt to delete the *IndividualSeller* but should be rejected.
- The ninth statement should delete the *Automobile* row successfully.
- The tenth statement should delete the *IndividualSeller* row successfully.
- The eleventh statement should delete the *Dealer* row successfully.

Problem 6.

Consider a table *Enroll* (*SID*, *CID*, *term*, *grade*). The following SQL statements are executed as a single transaction:

```
SELECT MIN(grade) FROM Enroll WHERE SID = 123;
SELECT MAX(grade) FROM Enroll WHERE SID = 123;
COMMIT;
```

Surprisingly, student 123's highest grade (as reported by the second statement) is lower than his lowest grade (as reported by the first statement).

(a) Suppose that only insertions are allowed on *Enroll*; both updates and deletions are disallowed. Could the anomaly described above happen if the transaction was run at the isolation level `READ COMMITTED`? Could it happen if the isolation level was `READ UNCOMMITTED`? Briefly explain why.

(b) Suppose that all kinds of modifications are allowed on *Enroll*. Could this anomaly happen if the transaction was run at the isolation level `READ COMMITTED`? Briefly explain why.