Written Assignment #5
Due Wednesday May 20

1. Consider the following four relations:

```
EmpDept(empID,deptNo)     // empID is key
EmpOffice(empID,office)   // empID is key
EmpPhone(empID,phone)     // empID is key
DeptMgr(deptNo,mgrID)     // deptNo is key
```

   and the following view that joins all four relations:

```
create view AllInfo as
   (select ED.empID, ED.deptNo, EO.office, EP.phone, DM.mgrID
    from EmpDept as ED, EmpOffice as EO, EmpPhone as EP, DeptMgr as DM
    where ED.empID = EO.empID
    and EO.empID = EP.empID
    and ED.deptNo = DM.deptNo)
```

   (a) Write a query to find the offices of all employees whose department manager has ID `123`. Do *not* use the view, and eliminate duplicate offices in your result.

   (b) Now write the same query using the view and not the base relations. Again, eliminate duplicate offices in your result.

   (c) Are the queries in parts (a) and (b) equivalent? If so, briefly explain why. If not, give a simple counterexample consisting of four relation instances, the view instance, and the two different query results.

2. Give an example of two simple relation schemas R_1 and R_2 where the `primary key` of R_1 is also a `foreign key` referencing the `primary key` of R_2, and the `primary key` of R_2 is also a `foreign key` referencing the `primary key` of R_1. You may choose any application domain you like, but the relations you choose and the pair of referential integrity constraints should make sense as representations of the real world.

3. Problems from the textbook:

   (a) Write the following as attribute-based `check` constraints: Parts (d) and (e) from Exercise 6.3.2 on pages 341–342.

   (b) Write the following as a tuple-based `check` constraint: Part (b) from Exercise 6.4.4 on page 347.

   (c) Write the following as general assertions: Parts (a), (c), and (d) from Exercise 6.4.3 on page 347.

   (d) For Exercise 6.4.3 part (c), rewrite the general assertion as one or more attribute-based or tuple-based `check` constraints such that it is impossible for the constraint to become violated, regardless of what database modifications occur.

4. In this problem you will show how to encode functional and multivalued dependencies as SQL constraints. In each case the constraint must be written so that it is impossible for the dependency to become violated, regardless of what database modifications occur.

   (a) Consider a relation $R(A, B)$ and the functional dependency $A \rightarrow B$.

      (1) Can the FD be declared as an attribute-based `check` constraint? If so, show how.

      (2) Can the FD be declared as a tuple-based `check` constraint? If so, show how.

      (3) Can the FD be declared as a general `assertion`? If so, show how.

   (b) Consider a relation $R(A, B, C)$ and the multivalued dependency $A \twoheadrightarrow B$.

      (1) Can the MVD be declared as an attribute-based `check` constraint? If so, show how.

      (2) Can the MVD be declared as a tuple-based `check` constraint? If so, show how.

      (3) Can the MVD be declared as a general `assertion`? If so, show how.

5. Views in SQL are *virtual*, meaning that they are not stored in the database but rather their definitions are used to translate queries referencing views into queries over base relations. One disadvantage of this approach is that views may effectively be computed over and over again if many queries reference the same view. An alternative approach is to *materialize* views: the contents of a view are computed and the result is stored in a database table, so that a reference to the view in a query can simply access the stored table. However, when contents of a base relation referenced in the view change, then the contents of the materialized view must be modified accordingly. For example, consider a base relation $R(A, B)$, where A and B are of type `integer` and A is a key for R. A materialized view V that contains those tuples of R satisfying $R.A > 5$ can be created as follows:

```
create table V (A int, B int)
```

   Initially, V is populated using the following SQL statement:

```
insert into V
   select * from R where A > 5
```

   Now when we refer to view/table V in queries we obtain the desired result.

   If an `insert` statement is executed on R, then V must be modified accordingly. This behavior can be implemented using a trigger:

```
create trigger VinsR
after insert on R
referencing new_table as NT
insert into V
   select * from NT where A > 5
```

   (a) Write another trigger `VdelR` to modify V after tuples are deleted from R.

   (b) Write another trigger `VupdR` to modify V after tuples in R are updated.

   (c) Write SQL statements and triggers to create, populate, and maintain a materialized view V that projects columns A and B from a base relation $R(A, B, C)$. You may assume that all three attributes are of type `integer` and that A is a key for R.

(d) (Optional – answer is long) Write SQL statements and triggers to create, populate, and maintain a materialized view V that is the natural join of base relations $R(A, B)$ and $S(B, C)$. You may assume that all four attributes are of type `integer` and that $R.A$ and $S.B$ are keys for R and S respectively.

6. Consider a relation `TA(name,class,salary)` and assume that `name` is a key.

(a) Consider the following trigger, specified using the `for each row` option:

```
create trigger 145Hardship
after update of class on TA
referencing old as O, new as N
when (N.class = 'CS145' and O.class <> 'CS145')
update TA
  set salary = 1.1 * salary
  where name = N.name
for each row
```

First describe in English what this trigger does. Then write an equivalent trigger that does not use the `for each row` option.

(b) (Tricky – takes some thought) Specify another trigger on the `TA` relation that does not use the `for each row` option such that an equivalent trigger *cannot* be written using the `for each row` option. (Assume that the equivalent trigger is not allowed to introduce temporary variables or tables.) In addition to specifying the trigger in SQL, describe in English what your trigger does.