View Modifications Using Triggers Exercises

You will create (virtual) views over the social-network database that was also used for the SQL Social-Network Query Exercises, and you will enable modifications to these views using triggers. A SQL file to set up the schema and data for the social-network database is downloadable here. The schema and data can be loaded as specified in the file into SQLite, MySQL, or PostgreSQL. However, currently only SQLite triggers and PostgreSQL rules can be used for these exercises. (MySQL instead supports automatic view modifications, as explored in the Automatic View Modifications Exercises.) See our quick guide for installing and using all three systems.

Schema:
Highschooler (ID, name, grade)
English: There is a high school student with unique *ID* and a given *first name* in a certain *grade*.

Friend (ID1, ID2)
English: The student with *ID1* is friends with the student with *ID2*. Friendship is mutual, so if (123, 456) is in the Friend table, so is (456, 123).

Likes (ID1, ID2)
English: The student with *ID1* likes the student with *ID2*. Liking someone is not necessarily mutual, so if (123, 456) is in the Likes table, there is no guarantee that (456, 123) is also present.

For your convenience, here is a graph showing the various connections between the people in our database. 9th graders are blue, 10th graders are green, 11th graders are yellow, and 12th graders are purple. Undirected black edges indicate friendships, and directed red edges indicate that one person likes another person.

In each exercise you are first asked to create a view, and then to use triggers (SQLite) or rules (PostreSQL) to enable one or more types of modifications to the view. Then you are given some modification commands to execute against the view. To verify correctness of your view-modification triggers, you are asked to compare your final database against our results.

1. Create a view called *JordanFriend*(name,grade) containing the names and grades of students with a friend named Jordan. Your view should initially contain (in some order):

Gabriel	9
Tiffany	9
Andrew	10
Kyle	12
Logan	12

Create a trigger (SQLite) or rule (PostgreSQL) that enables update commands to be executed on view *JordanFriend*. Updates should propagate to the *Highschooler* table under the assumption that (name,grade) pairs uniquely identify students. Do not allow updates that take the grade out of the 9-12 range, or that violate uniqueness of (name,grade) pairs; otherwise all updates should be permitted.

(a) Execute the following update command:

```
update JordanFriend set grade = grade + 2;
```

Compare your resulting view *JordanFriend* with ours (in Query Results at the bottom of the page).

(b) Then execute the following update commands:

```
update JordanFriend set name = 'Tiffany', grade = 10 where name = 'Gabriel';
update JordanFriend set name = 'Jordan' where name = 'Tiffany';
```

Compare your resulting view *JordanFriend* with ours. *(What do you think about the result?)*

Reload the original social-network database before beginning the next exercise.

2. Create a view called *OlderFriend*(ID1,name1,grade1,ID2,name2,grade2) containing the names and grades of friends who are at least two years apart in school, with name1/grade1 being the younger student. After reloading the original database, your view should initially contain (in some order):

1381	Tiffany	9	1247	Alexis	11
1709	Cassandra	9	1247	Alexis	11
1782	Andrew	10	1304	Jordan	12

Create triggers (SQLite) or rules (PostgreSQL) that enable deletions and insertions to be executed on view *OlderFriend*. For insertions, only permit new friendships that obey the restrictions of the view and do not create duplicates. Make sure to maintain the symmetric nature of the underlying *Friend* relation even though *OlderFriend* is not symmetric: a tuple (A,B) is in *Friend* if and only if (B,A) is also in *Friend*.

(a) Execute the following deletions:

```
delete from OlderFriend where name2 = 'Alexis';
delete from OlderFriend where ID1 = 1381;
```

Check the resulting database by writing SQL queries to compute the number of tuples in the *Friend* table and *OlderFriend* view. Compare your results against ours (in Query Results at the bottom of the page).

(b) Then execute the following insertions:

```
insert into OlderFriend values (1510, 'Jordan', 9, 1304, 'Jordan', 12);
insert into OlderFriend values (1510, 'Jordan', 9, 1468, 'Kris', 10);
insert into OlderFriend values (1510, 'Jordan', 9, 1468, 'Kris', 11);
insert into OlderFriend values (1510, 'John', 9, 1247, 'Alexis', 11);
insert into OlderFriend
    select H1.ID as ID1, H1.name as name1, H1.grade as grade1,
           H2.ID as ID2, H2.name as name2, H2.grade as grade2
    from Highschooler H1, Highschooler H2 where H1.grade >= 10;
```

Check the resulting database by writing SQL queries to compute the number of tuples in the *Friend* table and *OlderFriend* view. Compare your results against ours.

Hide Query Results

1.

(a) *In any order:*

Gabriel	9

Tiffany	11
Andrew	12
Kyle	12
Logan	12

(b) *In any order:*

Jordan	9
Jordan	10
Jordan	11
Cassandra	9
Andrew	12
Alexis	11
Kyle	12
Logan	12

2.

(a) *Friend* contains 36 tuples and *OlderFriend* contains 1 tuple

(b) *Friend* contains 68 tuples and *OlderFriend* contains 17 tuples