

Database Tutorial 4: Constraints, Triggers, and Views

Sample Solution - 30 November 2018

1. Consider an online bookshop which sometimes promotes books by displaying them on the front page of their website. Their web application uses a database created in PostgreSQL using the following statements:

```
CREATE TABLE Books (  
    id INTEGER PRIMARY KEY,  
    category TEXT,  
    price FLOAT CHECK (price>0),  
    promoted BOOLEAN DEFAULT True  
);
```

```
INSERT INTO Books (id, category, price) VALUES (1, 'Dictionary', 100);  
INSERT INTO Books (id, category, price) VALUES (2, 'Dictionary', 150);  
INSERT INTO Books (id, category, price) VALUES (3, 'Science', 120);  
INSERT INTO Books (id, category, price) VALUES (4, 'Science', 190);  
INSERT INTO Books (id, category, price) VALUES (5, 'Science', 320);
```

- a. Create a new VIEW called “PromotionSummary” which outputs 3 columns named “category”, “minprice” and “maxprice” containing the category name, minimum price of all promoted books and maximum price of all promoted books. A promoted book has its “promoted” attribute set to *True*.

```
CREATE VIEW PromotionSummary AS  
    SELECT category, MIN(price) AS minprice , MAX(price) AS maxprice  
    FROM Books  
    WHERE promoted  
    GROUP BY category;
```

- b. Create a trigger so that, when a tuple from the “PromotionSummary” view is deleted, all Books from the corresponding category have their “promoted” attribute set to *False*. E.g. if the entry in “PromotionSummary” for category “Novel” is deleted, all entries in “Books” with category “Novel” have their “promoted” attribute set to *False*.

```
CREATE OR REPLACE FUNCTION demoteBooks () RETURNS TRIGGER AS $$  
BEGIN  
    UPDATE Books SET promoted = False WHERE category = OLD.category;  
    RETURN OLD;  
END  
$$ LANGUAGE 'plpgsql';  
  
CREATE TRIGGER demoteBooksTrigger INSTEAD OF DELETE ON PromotionSummary  
FOR EACH ROW  
EXECUTE PROCEDURE demoteBooks();
```

2. Database integrity can be improved by many techniques:

- views: virtual tables that show useful information that would create redundancy if stored in the actual tables
- constraints: conditions on attribute values and tuples
- triggers: automated checks and actions performed on entire tables.

As a general rule, these methods should be applied in the above order: if a view can do the job, constraints are not needed, and if constraints can do the job, triggers are not needed. The task in this question is to show how to guarantee integrity in an exam question database. The attributes that are needed are the following:

- an Exam has a date and a course code, as well as a total number of points
- a Question belongs to a certain exam, has its own question number, a number of points, and a text (the question itself)

Your task is to give a database definition (tables, triggers, and views) that guarantees the following integrity conditions:

- a. A course can have maximally one exam on the same day.
- b. The number of points in an exam is the sum of the numbers of points in its questions.
- c. An exam question is uniquely determined by its number, together with the exam to which it belongs.
- d. An exam may not use the same question text twice.
- e. The number of questions in an exam may not exceed 10.
- f. The number of questions in an exam must be at least 5. (Hint: can be tricky!)

Write your answer as SQL code (except for question f) for tables, views, and triggers, marking with letters ((a), (b), etc.) which part of the code guarantees which property. You can get 2 points for each of the properties. Unnecessarily complicated answers (e.g. using a trigger when a constraint would be enough) may reduce points.

```
CREATE TABLE Courses (  
    code TEXT PRIMARY KEY  
);  
  
CREATE TABLE Exams (  
    course TEXT NOT NULL REFERENCES Courses(code),  
    date DATE NOT NULL,  
    PRIMARY KEY (course, date) (a)  
);
```

```

CREATE TABLE Questions (
    course TEXT,
    date DATE,
    qnumber INT,
    qtext TEXT NOT NULL,
    points FLOAT NOT NULL,
    FOREIGN KEY (course, date) REFERENCES Exams(course, date),
    PRIMARY KEY (course, date, qnumber),      (c)
    UNIQUE (course, date, qtext),             (d)
    CHECK (qnumber >= 1 AND qnumber <= 10)    (e)
);
CREATE View ExamPoints AS (                  (b)
    SELECT course, date, SUM(points) AS total
    FROM Questions
    GROUP BY (course, date)
);

```

Possible solutions to (f):

- Create a table PlannedQuestions to hold planned questions temporary (identical to Questions, but without a reference to Exams), create a trigger that after insert into Exams raises an error message if the number of planned questions *for that exam* is less than 5, otherwise moves the planned questions to the real questions table. After that you still need to create a trigger to prevent a questions being deleted (or moved using UPDATE) from an exam if the exam would have less than 5 questions after the operation.
- Create a view ValidExams that excludes exams with fewer than five questions (and maybe a separate view for invalid exams. Then when listing exams, only select from the view.
- Exams has 5*2 columns for five questions (and points), which are NOT NULL; extra questions are added separately. This could potentially be very messy.

3. In the year 2127, the first spaceship to colonize Mars carries 1337 colonists. When they arrive on the planet, they will build a city and live there. Following democratic principles, the spaceship captain, captain Picard, asks you to improve an SQL database to help with the voting process. The existing SQL database was created with the following statement:

```
CREATE TABLE Votes (  
    cityname TEXT PRIMARY KEY ,  
    votecount INT  
);
```

-- To add a vote , you can use either INSERT or UPDATE , as shown below:

```
INSERT INTO Votes (cityname , votecount ) VALUES( 'Mars City One' , 34 );
```

```
INSERT INTO Votes (cityname , votecount ) VALUES( 'New Gothenburg ' , 11);
```

```
INSERT INTO Votes (cityname , votecount ) VALUES( 'Picardia ' , 1);
```

```
UPDATE Votes SET votecount = votecount +3 WHERE cityname='New Gothenburg ';
```

- a. Create a new VIEW called “VoteSummary” which outputs 2 columns named “cityname” and “percentage” containing the cityname and percentage of votes cast for that cityname. The output is sorted according to the votecount, highest votecount first. After the example votes above, there would be 34 votes for “Mars City One” out of a total of 49 votes, so the top row of the “VoteSummary” VIEW would be ('Mars City One', 69.3878). There is no need to round of the percentage.

```
CREATE OR REPLACE VIEW VoteSummary AS  
    SELECT cityname, 100.0 * votecount / (SELECT SUM (votecount) FROM Votes ) AS  
    percentage  
    FROM Votes  
    ORDER BY percentage DESC;
```

- b. Create a trigger to update the “Votes” table, to keep track of how many colonists have not voted yet. This count will appear next to the special cityname “<not voted>”. In the example above, 49 votes have been cast out of 1337 possible votes. This means the trigger needs to create or update an entry with cityname “<not voted>” and votecount 1288 (= 1337 - 49). Keep in mind that you need to create this entry if it does not exist yet. There is no need for a trigger on DELETE.

```
CREATE OR REPLACE FUNCTION fixUnknownVotes () RETURNS TRIGGER AS $$  
BEGIN  
    IF NOT EXISTS (SELECT * FROM Votes WHERE cityname = '<not voted >')  
    THEN  
        INSERT INTO Votes (cityname, votecount) VALUES ('<not voted >', 0);  
    END IF;  
    UPDATE Votes SET votecount = 1337 -  
        (SELECT SUM(votecount) FROM Votes WHERE cityname <> '<not voted>')  
    WHERE cityname = '<not voted >';  
    RETURN NEW;  
END  
$$ LANGUAGE 'plpgsql';  
  
CREATE TRIGGER fixUnknownVotesTrigger AFTER INSERT OR UPDATE ON Votes  
    FOR EACH ROW WHEN (NEW. cityname <> '<not voted >')  
    EXECUTE PROCEDURE fixUnknownVotes ();
```

We can also write the function and the trigger differently (see the following).

Alternative solution (not using WHEN):

```
CREATE OR REPLACE FUNCTION updateNotVoted() RETURNS TRIGGER AS $$
BEGIN
    IF NEW.cityname <> '<not voted>'
    THEN
        IF NOT EXISTS (SELECT * FROM Votes WHERE cityname = '<not voted>')
        THEN
            INSERT INTO Votes (cityname, votecount) VALUES ('<not voted>', 0);
        END IF;

        UPDATE Votes SET votecount = 1337-(SELECT SUM(votecount) FROM Votes
        WHERE cityname <> '<not voted>') WHERE cityname = '<not voted>';
        RETURN NEW;
    ELSE
        RETURN NULL;
    END IF;
END
$$ LANGUAGE 'plpgsql';

CREATE TRIGGER updateNotVotedTrigger
    AFTER INSERT OR UPDATE ON Votes
    FOR EACH ROW
    EXECUTE PROCEDURE updateNotVoted();
```

Note that creating a View would be a much better solution to this problem! Just make the view show the actual votes table, and add a single row for the <not voted> using UNION.