Databases - Exercise 4:
Views, Constraints, Triggers

Sample solution

29 November 2019

1 Constraints and Triggers

Consider the following CREATE TABLE statements.

```
CREATE TABLE Artists(
  name TEXT PRIMARY KEY);

CREATE TABLE Songs(
  title TEXT PRIMARY KEY,
  length INT NOT NULL,
  artistName TEXT NOT NULL,
  FOREIGN KEY (artistName) REFERENCES Artists(name),
  CONSTRAINT SongTitleAndArtistNameIsUnique UNIQUE (title, artistName)
);

CREATE TABLE Playlists (
  id TEXT PRIMARY KEY,
  name TEXT,
  owner TEXT);

CREATE TABLE PlaylistSongs (
  playlist TEXT REFERENCES Playlists(id),
  song TEXT,
  artist TEXT,
  position INTEGER,
  FOREIGN KEY (song,artist) REFERENCES Songs(title,artistName),
  PRIMARY KEY (playlist,position)
);
```

1.1 Constraints

Which of the following properties are guaranteed by the given constraints?

1. Every song in the playlist exists.
2. All playlists of one and the same owner have different names.
3. All positions in a given playlist are unique.
4. The positions can be listed in order 1, 2, 3, ... with no numbers missing.

Answer as follows:

- If a property is guaranteed, say by which constraints exactly.
- If a property is not guaranteed, give a counterexample.

Solution

1. Every song in the playlist exists.
   Yes. FOREIGN KEY (song,artist) REFERENCES Songs(title,artistName) in CREATE TABLE statement for PlaylistSongs makes sure that the song can be added only if it exists in Table Songs.
2. All playlists of one and the same owner have different names.
   No. Same owner can have more than one playlists with same name, since the key is only the id. For example, the following would be accepted:
   - PL001, jazz1, Joe
   - PL002, jazz1, Joe
3. All positions in a given playlist are unique.
   Yes. The PRIMARY KEY (playlist,position) in CREATE TABLE statement for PlaylistSongs makes sure of this.
4. The positions can be listed in order 1, 2, 3, ... with no numbers missing.
   No. The following would be accepted: 2, 5, 9, ...

1.2 Views

Create a view that, for a playlist with id M123, shows the contents of the playlist with the following layout:

```
 position | song               | artist       | length
----------+--------------------+--------------+--------
        1 | Dancing Quenn      | ABBA         | 232
        2 | Too late for love  | John Lundvik | 187
```

Solution

```
CREATE VIEW PlaylistM123 AS (
  SELECT position, song, artist, length
  FROM PlaylistSongs, Songs
  WHERE playlist = 'M123'
  AND song = Songs.title
  AND artist = Songs.artistName
  ORDER BY position
);
```

1.3 Triggers

Create a trigger that enables directly building the playlist M123 via the view defined in the previous question. The behaviour should be as follows:

- Insertion of (position, song, artist, length) in M123 means an insert in PlaylistSongs such that
  - the song and artist are inserted as they are given by the user
  - the length is ignored (even if it contradicts the Songs table)
  - the position given by the user is respected, at the same time maintaining the sequential order 1, 2, 3, ... of the playlist

Maintaining the sequential order implies the following: assume that the positions in the old playlist are 1,2,3,4. Then

- if the user inserts in the next position, 5, then 5 is also used as the position of the row inserted to the table
- if it is larger, say 8, then the position of the row inserted to the table is still 5

- if it is smaller, say 3, then the position of the row inserted is 3, but the positions of the old rows 3 and 4 are changed to 4 and 5, respectively

Note: You may notice the problem that, while the trigger is running, the unicity of positions is temporarily violated. But you can ignore the complications with this: just make sure that, when the trigger has finished its work, all positions are unique.

Solution

```
CREATE FUNCTION insertPlaylistFunction()
    RETURNS TRIGGER AS $$
BEGIN
  IF (NEW.position > (SELECT MAX(position) FROM PlaylistSongs
        WHERE playlist = 'M123'))
    THEN INSERT INTO PlaylistSongs VALUES ('M123', NEW.song,
        NEW.artist, 1+(SELECT MAX(position) FROM
        PlaylistSongs WHERE playlist = 'M123'));
  ELSE
    UPDATE PlaylistSongs
        SET position = position + 1
        WHERE position >= NEW.position AND playlist='M123';
    INSERT INTO PlaylistSongs VALUES ('M123', NEW.song,
        NEW.artist, NEW.position);
  END IF;
  RETURN NEW;
END
$$ LANGUAGE 'plpgsql';

CREATE TRIGGER insertPlaylist
  INSTEAD OF INSERT ON PlaylistM123
  FOR EACH ROW
  EXECUTE PROCEDURE insertPlaylistFunction();
```

2 Views, constraints, and triggers again

Database integrity can be improved by several techniques:

- Views: virtual tables that show useful information that would create redundancy if stored in the actual tables

- SQL Constraints: conditions on attribute values and tuples
- Triggers (and assertions): 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 implement a database for a cell phone company.

You are allowed to use any SQL features we have covered in the course. While the description below gives requirements for what should be in the database, you are allowed to divide it across as many tables and views as you need to. Points will be deducted if your solution uses a trigger where a constraint or view would suffice, or if your solution is drastically over-complicated.

For triggers, it is enough to specify which actions and tables it applies to, and PL/(pg)SQL pseudo-code of the function it executes.

Your task: The database contains Customers and Subscriptions. Each customer can have any number of subscriptions. Below are values that should be in the database:

- A Customer has a unique id number and a name, a monthly billing and a Boolean indicating if it is a private customer (true meaning it is a private customer).
- A Subscription belongs to a customer and has a unique phone number, a plan, a monthly fee and a balance.

Implement the following additional constraints in your design.
Put letters in the margin of your code indicating where each constraint is implemented (possibly the same letter in several places):

a Each plan must be one of 'prepaid', 'flatrate' or 'corporate'.

b The balance value must be 0 if the plan is not 'prepaid'.

c Private customers cannot have 'corporate' plans (but non-private customers may still have any plans including but not limited to 'corporate').

d The monthly billing of a customer must be the sum of the fees of all that customers numbers, and all fees must be non-negative.

e If a customer is deleted, its connected subscriptions should be deleted automatically.

f If the last subscription belonging to a customer is deleted, the customer should be deleted automatically.

Solution

```
CREATE TABLE Customers (
  id INT PRIMARY KEY,
  name TEXT,
  isPrivate BOOLEAN,
  UNIQUE (id, isPrivate)  -- because of being referenced
);

CREATE TABLE Subscriptions (
  phoneNumber TEXT PRIMARY KEY,
  customer INT,
  isPrivate BOOLEAN,      -- added because of c
  plan TEXT,
  fee INT,
  balance INT,
  FOREIGN KEY (customer, isPrivate)
    REFERENCES Customers(id, isPrivate)
    ON DELETE CASCADE,                             -- e
  CHECK (plan IN ('prepaid', 'corporate', 'flatrate')), -- a
  CHECK (plan='prepaid' OR balance=0),             -- b
  CHECK (plan!='corporate' OR NOT isPrivate),      -- c
  CHECK (fee >= 0)                                 -- d
);

CREATE VIEW CustomerView AS                        -- d
  SELECT id, name, Customers.isPrivate, SUM(fee)
  FROM Customers JOIN Subscriptions ON id=customer
  GROUP BY id, name, Customers.isPrivate;

CREATE FUNCTION deleteEmpty() RETURNS trigger AS $$
BEGIN
  IF NOT EXISTS (SELECT * FROM Subscriptions
      WHERE customer = OLD.customer)
    THEN DELETE FROM Customers WHERE id=OLD.customer;
  END IF;
  RETURN OLD;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER deleteEmpty                         -- f
  AFTER DELETE ON Subscriptions
  FOR EACH ROW
  EXECUTE PROCEDURE deleteEmpty();
```

3 Troubleshooting

Rotten Potatoes is a company with a website where users can review movies by rating and commenting on them. The company is requesting your services as a consultant since its database is not behaving as expected. Your task is to find why that is and propose how to fix it.

Domain description

- Users have a login which identifies them along with an unique email. Users can pay a fee to become premium, and their subscriptions have an expiration date.
- Movies have a title, year of release, and a genre that can be either action, sci-fi, comedy, or drama. Since movies can share tiles, they are disambiguated by their year of release.
- Users review movies by rating them with a point system from 1 to 5 stars; additionally, premium users can comment on their reviews.
- A movie score is determined by the average of its ratings.
- To keep users up to date, Rotten Potatoes provides a list of "certified crisp" movies, which are the movies whose score is above 4.

Questions

1. Is their implementation *addressing* all the constraints described in the domain? If so, highlight where, otherwise, mention what is missing.

   Solution

   YES. line 3: login identifies users, line 5: users' emails are unique, line 18: movies are uniquely identified by their year of release, line 19: movies genre is limited to the specified categories, line 29: movies rates go from 1 to 5 stars, line 33: only premium users can add comments, lines 47-50: movies' score is computed as the average of its rates, lines 58-79: only movies with score over 4 are added to certified crisp

2. After inserting a couple of reviews the company has noticed some movies listed as "certified crisp" whose average is less than 4. Why is this happening? what needs to be changed/added/removed?

   Solution

   Function `insert_review` checks if the `newScore` is over 4 to insert the movie en `CertifiedCrisp`, however, it never checks if `newScore` is less or equal than 4 and the movie already has been classified as crisp, if that is the case, it should be deleted from `CertifiedCrisp`.
   To solve this issue we can add the following `ELSE IF` statement between lines 75-76

```
ELSE IF (EXISTS (SELECT * FROM CertifiedCrisp
          WHERE (title,year) = (NEW.movie, NEW.year))
         AND (newScore <= 4))
     THEN
         DELETE FROM CertifiedCrisp WHERE title = NEW.movie
                AND year = NEW.year;
     END IF;
```

3. To clean the database from fake accounts, the company have removed all reviews posted by shady users; however, "certified crisp" remain unchanged. What went wrong? how can it be fixed?

Solution

Function `delete_review` never updates `CertifiedCrisp` accordingly, to do so, we need to add a variable for the new score and perform the same checks that where done in `insert_review`—including what we added in the previous question.

```
CREATE OR REPLACE FUNCTION delete_review ()
  RETURNS trigger AS $$
DECLARE newScore FLOAT; --NEW
BEGIN
  ...
  newScore := (SELECT score FROM MoviesScores
               WHERE (movie,year) = (OLD.movie, OLD.year));

  IF (NOT EXISTS (SELECT * FROM CertifiedCrisp
          WHERE (title,year) = (OLD.movie, OLD.year))
      AND (newScore > 4))
  THEN INSERT INTO CertifiedCrisp
              VALUES (OLD.movie, OLD.year);
  ELSE IF (EXISTS (SELECT * FROM CertifiedCrisp
           WHERE (title,year) = (OLD.movie, OLD.year))
          AND (newScore <= 4))
       THEN DELETE FROM CertifiedCrisp
            WHERE title = OLD.movie AND year = OLD.year;
       END IF;
  END IF;
  RETURN OLD;
END;
```

4. Is there redundancy in the information that is being stored? If so, where is it happening? and how can this be improved?

Solution

YES. There is redundancy since table `CertifiedCrisp` can be easily modeled as a view, which indicates that we are storing information that is already in the database.

```
CREATE VIEW CertifiedCrip AS
    SELECT movie, year FROM MoviesScores
    WHERE (score > 4);
```

Once the table is replaced by the previous view, functions `insert_review` and `delete_review` can be simplified removing all the logic that was taking care of updating table `CertifiedCrisp`.

Implementation

```sql
-- Users' basic information
CREATE TABLE Users (
    login TEXT PRIMARY KEY,
    email TEXT NOT NULL,
    UNIQUE (email));

-- Premium users are an specialization of users
CREATE TABLE PremiumUsers (
    usr             TEXT NOT NULL REFERENCES Users,
    expiration_date DATE NOT NULL,
    PRIMARY KEY (usr));

-- Movies' basic information
CREATE TABLE Movies (
    title       TEXT    NOT NULL,
    year        INTEGER NOT NULL,
    genre       TEXT    NOT NULL,
    PRIMARY KEY (title, year),
    CHECK (genre IN ('Action', 'Sci-fi', 'Comedy', 'Drama')));

-- All users can rate a movie
CREATE TABLE Ratings (
    usr   TEXT    NOT NULL REFERENCES Users,
    title TEXT    NOT NULL,
    year  INTEGER NOT NULL,
    rate  INTEGER NOT NULL,
    PRIMARY KEY (usr, title, year),
    FOREIGN KEY (title, year) REFERENCES Movies,
    CHECK (rate IN (1,2,3,4,5)) -- Point system);

-- Only premium users are allowed to comment
CREATE TABLE Comments (
    usr     CHAR(5) NOT NULL REFERENCES PremiumUsers,
    title   TEXT    NOT NULL,
    year    INTEGER NOT NULL,
    comment TEXT    NOT NULL,
    PRIMARY KEY (usr, title, year),
    FOREIGN KEY (title, year) REFERENCES Movies);

-- Movies with score above 4.0
CREATE TABLE CertifiedCrisp (
    title   TEXT    NOT NULL,
    year    INTEGER NOT NULL,
    PRIMARY KEY (title, year),
    FOREIGN KEY (title, year) REFERENCES Movies);

```

```
CREATE VIEW MoviesScores AS
    SELECT title AS movie, year, COALESCE (AVG(rate),0) AS score
    FROM Movies NATURAL JOIN Ratings
    GROUP BY (title , year);

CREATE VIEW Reviews AS
    SELECT usr, title AS movie, year, rate, comment
    FROM MOVIES
    NATURAL LEFT JOIN Ratings
    NATURAL LEFT JOIN Comments;

CREATE FUNCTION insert_review () RETURNS trigger AS $$
DECLARE newScore FLOAT;
BEGIN
    INSERT INTO Ratings VALUES (NEW.usr, NEW.movie, NEW.year, NEW.rate);

    IF (EXISTS ( SELECT usr FROM PremiumUsers WHERE usr = NEW.usr)
        AND NEW.comment IS NOT NULL)
    THEN INSERT INTO Comments
                VALUES (NEW.usr, NEW.movie, NEW.year, NEW.comment);
    END IF;

    newScore := (SELECT score FROM MoviesScores
                 WHERE (movie,year) = (NEW.movie, NEW.year));

    IF (NOT EXISTS (SELECT * FROM CertifiedCrisp
                    WHERE (title,year) = (NEW.movie, NEW.year))
        AND (newScore > 4))
    THEN INSERT INTO CertifiedCrisp VALUES (NEW.movie, NEW.year);
    END IF;
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE FUNCTION delete_review () RETURNS trigger AS $$
BEGIN
    DELETE FROM Ratings WHERE usr = OLD.usr AND title = OLD.movie
           AND year = OLD.year;

    IF (EXISTS (SELECT usr FROM PremiumUsers WHERE usr = OLD.usr))
    THEN
        DELETE FROM Comments WHERE usr = OLD.usr AND title = OLD.movie
               AND year = OLD.year;
    END IF;
    RETURN OLD;
END;
$$ LANGUAGE plpgsql;

```

```
CREATE TRIGGER InsertReview
  INSTEAD OF INSERT ON Reviews
    FOR EACH ROW EXECUTE PROCEDURE insert_review();

CREATE TRIGGER DeleteReview
  INSTEAD OF DELETE ON Reviews
    FOR EACH ROW EXECUTE PROCEDURE delete_review();
```

The website only allows to insert and delete reviews through Reviews view so that users' premium subscriptions are checked, and "certified crisp" is updated accordingly.