Databases (TDA357, DIT620)

Exercises 4 : CONSTRAINTS & TRIGGERS Solutions 21-Feb-2018

Exercise 1: Create the tables Employee & Department with the following constraints and attributes respectively.

Employee
1. Id → Primary Key
2. Name → NOT NULL
3. DeptId → Foreign Key refering Dept(Dept_Id)
4. Salary→ Check Salary between 22000 to 50000
5. Insurance_opted→ Default value of Yes

Department
1. Dept_Id→ Unique, NotNull
2. Name→ Unique

Solution :

```
CREATE TABLE EMPLOYEE(ID INT PRIMARY KEY,
NAME TEXT NOT NULL,
deptid INT REFERENCES DEPARTMENT(DEPT_ID),
 SALARY INT CHECK(SALARY > 22000 AND SALARY < 50000),
 INSURANCE_OPTED TEXT DEFAULT 'YES');

 CREATE TABLE DEPARTMENT(DEPT_ID INT NOT NULL UNIQUE,
 NAME TEXT UNIQUE);
```

Exercise 2: Constraints and triggers are techniques that can be used to control properties of databases. As a rule of thumb, constraints are the simplest solution, but not always possible. In the following cases, use either constraints (added to a table) or triggers (on a table), whichever you find to be the simplest solution.

1. Assume you want to create a table that should have at most one row. How can you guarantee this? Write an example CREATE TABLE statement, together with a trigger if you need one.
2. Assume you have a table Teachers(name,phoneNumber) that lists the teachers currently in charge for some class. Assume the table already has some rows. How can you guarantee that this

table will never become empty i.e. never have 0 rows? Show the exact CREATE TABLE statement and/or trigger.
3.
Consider a table with the schema Distances(fromCity,toCity,distance) that stores distances between cities. Since the distance from Y to X is always the same as the distance from X to Y, it would be redundant to store them both. How can you guarantee that the table never stores the distance from Y to X if it already has the distance from X to Y ? Show the exact CREATE TABLE statement and/or trigger.

1.

```
CREATE or replace FUNCTION COUNT2() RETURNS TRIGGER AS $$
BEGIN
  IF ((SELECT COUNT(theRow) FROM AtMostOneROWS) >1)
          THEN RAISE EXCEPTION 'Cannot Insert another row' ;
  END IF ;
RETURN NULL;
END
$$ LANGUAGE 'plpgsql' ;

CREATE TRIGGER Atmost
  after INSERT ON AtMostOneROWS
  FOR EACH ROW
  EXECUTE PROCEDURE COUNT2() ;
```

--**ALTERNATE SOLUTION IF YOU KNOW THE EXACT POSSIBLE VALUE THAT CAN BE IN THE TABLE**

```
CREATE TABLE AtMostOnes(
  theRow INT PRIMARY KEY CHECK(theRow = 789)
) ;
```

2.

```
CREATE OR REPLACE FUNCTION notNoTeachers() RETURNS TRIGGER AS $$
BEGIN
RAISE NOTICE 'FROM TRIGGER FUNCTION';
IF ((SELECT COUNT(name) FROM Teachers) < 1)
          THEN
                  RAISE EXCEPTION 'no teacher left' ;
END IF ;
RETURN NULL ;
END
$$ LANGUAGE 'plpgsql' ;
```

```
CREATE TRIGGER guaranteeTeachers
  AFTER DELETE ON Teachers
  FOR EACH ROW
  EXECUTE PROCEDURE notNoTeachers() ;
```

3.

```
CREATE TABLE Distances (
  fromCity TEXT,
  toCity TEXT,
  distance INT,
  CONSTRAINT only_one_direction CHECK (fromCity < toCity),
  CONSTRAINT UNIQ UNIQUE(fromCity,toCity)
 ) ;
```

--ALTERNATE SOLUTION USING TRIGGERS

```
CREATE TABLE Distances (
  fromCity TEXT,
  toCity TEXT,
  distance INT)


CREATE OR REPLACE FUNCTION ensureCity() RETURNS TRIGGER AS $$
DECLARE
COUNTS INT;
BEGIN
COUNTS := (SELECT COUNT(*) FROM Distances where fromCity = NEW.toCity AND
toCity = NEW.fromCity);
IF(COUNTS > 0)
   THEN RAISE EXCEPTION 'Cannot Insert Duplicate row' ;
END IF;
RETURN NEW;
END
 $$ LANGUAGE 'plpgsql';

CREATE TRIGGER ensureDirection BEFORE INSERT ON Distances FOR EACH ROW
EXECUTE PROCEDURE ensureCity();
```

Exercise 3: Consider an online bookshop which sometimes promotes books by displaying them on the front page of their website. Create the table with the following constraints.

- Id → NOT NULL AND UNIQUE
- Category → Text
- Price → Float && Price >0
- Promoted → Default value of True

Their web application uses a database created in PostgreSQL using the following statements:

```
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.

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.

Solution :

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


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

CREATE OR REPLACE FUNCTION demoteBooks() RETURNS TRIGGER AS $$
BEGIN
UPDATE Books SET promoted = False WHERE category = OLD.category;
RETURN NEW;
END
 $$ LANGUAGE 'plpgsql';

CREATE TRIGGER demoteBooksTrigger INSTEAD OF DELETE ON PromotionSummary FOR
EACH ROW
EXECUTE PROCEDURE demoteBooks();
```

--Alternate Solution

demoteBooks() → UPDATE Books SET promoted = False WHERE category = OLD. category;
CREATE TRIGGER demoteBooksTrigger INSTEAD OF DELETE ON PromotionSummary
FOR EACH ROW
EXECUTE PROCEDURE demoteBooks()

Exercise 4: 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.

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 “”. 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 “” 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.

Solution :

```
CREATE or replace VIEW VoteSummary AS
 SELECT cityname, 100.0 * votecount / (SELECT SUM(votecount) FROM Votes) as percentage
 from Votes;

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 NULL;
END
$$ LANGUAGE 'plpgsql';

CREATE TRIGGER fixUnknownVotesTrigger AFTER INSERT OR UPDATE ON Votes
FOR EACH ROW
WHEN (NEW.cityname <> '<not_voted>')
EXECUTE PROCEDURE fixUnknownVotes();
```

Exercise 5: In this exercise, we are creating the following database for a restaurant:
Tables(number,seats)
Bookings(namr,time,nbpeople,table)
table→Tables.number

For the sake of simplicity, we assume that the database only needs to hold bookings for the current day and that bookings are always done on the hour (i.e. time is an integer between 0 and 23).
Finally, tables are always booked for two hours. This means that if table 1 is booked at 19.00, it can't be booked at 20.00 but it can be booked again at 21.00.

1. Write a view that lists the times at which tables are blocked by a booking. In the example above, where table 1 is booked at 19.00, the view should contain the following rows:

table	time
1	18
1	19
1	20

2. Write a trigger on the table Bookings that automatically assign a table to new rows if none is specified. The assigned table should respect the following rules:

- it should be free for the duration of the booking.
- it should be big enough for the number of people in the party
- it should be the smallest possible table to accommodate this number of people

--Solution 1¶

```
CREATE TABLE tables (number int, seats int);
 alter table tables add primary key(number);

CREATE TABLE bookings(name varchar(30), time int check(time<23), nbpeople int, table_no int,
constraint rela FOREIGN KEY(table_no) references tables(number));
CREATE OR REPLACE VIEW BlockedTables AS
SELECT table_no, time - 1 FROM Bookings WHERE time > 0
UNION
SELECT table_no, time FROM Bookings
UNION
SELECT table_no, time + 1 FROM Bookings WHERE time < 23;
```

--**Solution 2¶**

```
CREATE OR REPLACE FUNCTION assign_table() RETURNS TRIGGER AS $$
BEGIN
  NEW.table_no :=(
        WITH possible_tables AS (
        SELECT number, seats
        FROM tables
        WHERE (number, new.time) NOT IN (SELECT * FROM BlockedTables)
        AND seats >= new.nbpeople
        )
        SELECT MIN(number)
        FROM possible_tables
        WHERE seats = (SELECT MIN(seats) FROM possible_tables));


        IF (NEW.table_no IS NULL) THEN
        RAISE EXCEPTION 'No table available';
        END IF;


  RETURN NEW;
END
$$ LANGUAGE 'plpgsql';

CREATE TRIGGER AssignTable BEFORE INSERT ON Bookings
  FOR EACH ROW
  WHEN (NEW.table_no IS NULL)
  EXECUTE PROCEDURE assign_table();
```

Some Useful commands :

1 To Disable a trigger on a table : ALTER TABLE AtMostOnes DISABLE TRIGGER Atmost;
2 To disable all triggers on a table : ALTER TABLE AtMostOnes DISABLE TRIGGER ALL;
3.To drop a trigger on a table : DROP TRIGGER Atmost ON employees
4.To view the list of triggers : select * from information_schema.triggers
5.To view the list of functions : \df
6.To see the source code of a function/program : SELECT prosrc FROM pg_proc WHERE proname = 'count1'