Laboratory work 8

Student: Almat Begaidarov

Instructor: Aibek Kuralbaev

Assistant: Bermagambet Duisek

1. Create a function that:
a. Increments given values by 1 and returns it.
b. Returns sum of 2 numbers.
c. Returns true or false if numbers are divisible by 2.
d. Checks some password for validity.
e. Returns two outputs, but has one input.

```
-- 1
-- a
CREATE FUNCTION incr(inout n int)
    AS $$
    BEGIN
        n = n + 1;
    END; $$
LANGUAGE plpgsql;

SELECT incr(5);

-- b
CREATE FUNCTION sum2(in a int, in b int, out sm int)
    AS $$
    BEGIN
        sm = a + b;
    END; $$
LANGUAGE plpgsql;

SELECT sum2(2, 3);

-- c
CREATE FUNCTION isdiv2(n int)
    RETURNS bool
    AS $$
    BEGIN
        if(n % 2 = 0) then return true;
        else return false;
        end if;
    END; $$
LANGUAGE plpgsql;

SELECT isdiv2(10);

-- d
CREATE FUNCTION check_password(s varchar)
    RETURNS bool
    AS $$
    BEGIN
        if(s SIMILAR TO '[A-Za-z0-9]+[\.]*[A-Za-z0-9]+' AND length(s) >= 8) then
return true;
        else return false;
        end if;
    END; $$
LANGUAGE plpgsql;

SELECT check_password('abcd.1234')

-- e
CREATE FUNCTION func_e(in r numeric, out area numeric, out length numeric)
    AS $$
    BEGIN
```

```
        area = 3.14 * r * r;
        length = 2 * 3.14 * r;
    END; $$
LANGUAGE plpgsql;
```

2. Create a trigger that:
a. Return timestamp of the occured action within the database.
b. Computes the age of a person when persons' date of birth is inserted.
c. Adds 12% tax on the price of the inserted item.
d. Prevents deletion of any row from only one table.
e. Launches functions 1.d and 1.e.

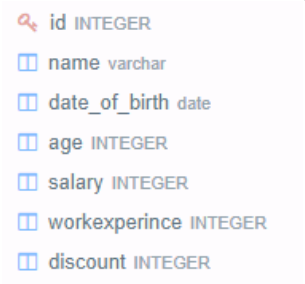
```
-- 2
-- a
CREATE TABLE games(
    id SERIAL PRIMARY KEY ,
    title VARCHAR(50) NOT NULL,
    genre VARCHAR(50) NOT NULL,
    changed timestamp
);

CREATE FUNCTION changes()
    RETURNS TRIGGER
    AS $$
    BEGIN
    new.changed= now();
    RETURN new;
    END; $$
LANGUAGE plpgsql;

CREATE TRIGGER game_changed
    BEFORE INSERT OR UPDATE on games
    FOR EACH ROW EXECUTE PROCEDURE changes();

INSERT INTO games(title, genre) values ('Warcraft', 'RPG');
INSERT INTO games(title, genre) values ('CSGO', 'FPS');
INSERT INTO games(title, genre) values ('Outlast', 'Horror');

SELECT * FROM games;

-- b
CREATE TABLE people(
    id SERIAL PRIMARY KEY,
    name VARCHAR(50) NOT NULL,
    birth_date date,
    age int
);

CREATE FUNCTION def_age()
    RETURNS TRIGGER
    AS $$
    BEGIN
        new.age = extract(year FROM current_date) - extract(year FROM new.birth_date);
        RETURN new;
    END; $$
LANGUAGE plpgsql;

CREATE TRIGGER comp_age
    BEFORE INSERT OR UPDATE on people
    FOR EACH ROW EXECUTE PROCEDURE def_age();

SELECT * FROM people;

INSERT INTO people(name, birth_date) VALUES ('Almat', '2002-03-23');
INSERT INTO people(name, birth_date) VALUES ('Manat', '1997-05-29');

-- c
```

```sql
CREATE TABLE products(
    id SERIAL PRIMARY KEY,
    name VARCHAR(50),
    initial_price NUMERIC,
    eventual_price NUMERIC
);

CREATE FUNCTION tax()
    RETURNS TRIGGER
    AS $$
    BEGIN
        new.eventual_price = new.initial_price * 1.12;
        RETURN new;
    END; $$
LANGUAGE plpgsql;

CREATE TRIGGER final_price
    BEFORE INSERT OR UPDATE on products
    FOR EACH ROW EXECUTE PROCEDURE tax();

INSERT INTO products(name, initial_price) VALUES ('ASUS TUF', 1000);
INSERT INTO products(name, initial_price) VALUES ('Macbook', 2500);
INSERT INTO products(name, initial_price) VALUES ('Lenovo Thinkbook', 600);

SELECT * FROM products;

-- d
CREATE FUNCTION proh_del()
    RETURNS TRIGGER
    AS $$
    BEGIN
        RAISE EXCEPTION 'You are unable to delete data from this table';
    END; $$
LANGUAGE plpgsql;

CREATE TRIGGER no_del
    BEFORE DELETE on products
    FOR EACH ROW EXECUTE PROCEDURE proh_del();

DELETE FROM products
WHERE id = 1;

-- e
CREATE TABLE passwords_val(
    id SERIAL PRIMARY KEY,
    user_name VARCHAR(50) NOT NULL,
    password VARCHAR(50) NOT NULL,
    isval bool
);
CREATE FUNCTION check_passwords()
    RETURNS TRIGGER
    AS $$
    BEGIN
        if(new.password SIMILAR TO '[A-Za-z0-9]+[\.]*[A-Za-z0-9]+' AND
length(new.password) >= 8) then new.isval = true;
        else return new.isval = false;
        end if;
        RETURN new;
    END; $$
LANGUAGE plpgsql;

CREATE TRIGGER password_trigger
    BEFORE INSERT OR UPDATE on passwords_val
    FOR EACH ROW EXECUTE PROCEDURE check_passwords();

INSERT INTO passwords_val(user_name, password) VALUES ('user1', '12345abcd');
SELECT * FROM passwords_val;
```

3.What is the difference between procedure and function?

```
-- 3
-- Procedure
-- Procedures will not return the value
-- Procedures always executes as PL SQL statement
-- It does not contain return clause in header section
-- We can pass the values using IN OUT IN OUT parameters
-- Procedures can not be executed in Select statement
-- Procedures cannot be called from a Function

-- Function
-- Functions must return the value. When you are writing functions make sure that you
can write the return statement
-- Functions executes as part of expression
-- It must contain return clause in header
-- Function must return a single value
-- Functions can execute or call using select statement but it must not contain Out or
IN OUT parameters
-- Functions can be called from Procedure
```

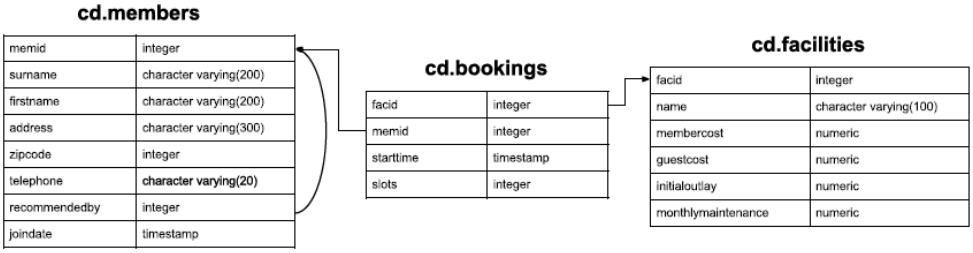
4.Create procedures that:
a) Increases salary by 10% for every 2 years of work experience and provides 10% discount and after 5 years adds 1% to the discount.
b) After reaching 40 years, increase salary by 15%. If work experience is more than 8 years, increase salary for 15% of the already increased value for work experience and provide a constant 20% discount.
Consider the following schema for the task4:




```
-- 4
CREATE TABLE employees(
    id SERIAL PRIMARY KEY,
    name VARCHAR,
    date_of_birth date,
    age int,
    salary int,
    work_exp int,
    discount numeric
);

INSERT INTO employees(name, date_of_birth, age, salary, work_exp, discount) VALUES
('Michael', '1987-07-09' , 34, 2000, 9, 0);
INSERT INTO employees(name, date_of_birth, age, salary, work_exp, discount) VALUES
('Alex', '1989-07-09' , 32, 1700, 7, 1);

-- a
CREATE PROCEDURE incr1()
      AS $$
      BEGIN
          UPDATE employees SET salary = salary * 1.1
            WHERE work_exp >= 2;
          UPDATE employees SET discount = discount * 1.1
            WHERE work_exp >= 2;
          UPDATE employees SET discount = discount * 1.01
            WHERE work_exp >= 5;
      END; $$
LANGUAGE plpgsql;
```

```
call incr1();

SELECT * FROM employees;

-- b
CREATE PROCEDURE incr2()
      AS $$
      BEGIN
          UPDATE employees SET salary = salary * 1.15
            WHERE age >= 40;
          UPDATE employees SET salary = salary * 1.15
            WHERE work_exp >= 8;
          UPDATE employees SET discount = discount * 1.20
            WHERE work_exp >= 8;
      END; $$
LANGUAGE plpgsql;

call incr2();

SELECT * FROM employees;
```

5. Produce a CTE that can return the upward recommendation chain for any member. You should be able to `select recommender from recommenders where member=x`. Demonstrate it by getting the chains for members 12 and 22. Results table should have member and recommender, ordered by member ascending, recommender descending.
Consider the following schema for the task5:




```
-- 5
CREATE TABLE members(
    memid int,
    surname VARCHAR(200),
    firstname VARCHAR(200),
    address VARCHAR(300),
    zipcode int,
    telephone VARCHAR(20),
    recommendedby int,
    joindate timestamp
);

CREATE TABLE bookings(
    facid int,
    memid int,
    starttime timestamp,
    slots int
);

CREATE TABLE facilities(
    facid int,
    name VARCHAR(100),
    membercost numeric,
    guestcost numeric,
    initialoutlay numeric,
    monthlymaintenance numeric
);
```

```
WITH RECURSIVE recommenders(recommender, member) AS
    (SELECT recommendedby, memid FROM members UNION ALL
    SELECT m.recommendedby, r.member FROM recommenders r INNER JOIN members m on
m.memid = r.recommender)
SELECT r.member, r.recommender, m.firstname, m.surname FROM recommenders r INNER JOIN
members m on r.recommender = m.memid
    WHERE r.member in (12, 22)
    ORDER BY r.member ASC, r.recommender DESC;
```