PostgreSQL

PostgreSQL est un système de gestion de base de données relationnelle et objet. C'est un outil libre disponible selon les termes d'une licence de type BSD. Ce système est comparable à d'autres systèmes de gestion de base de données, qu'ils soient libres, ou propriétaires.

Trigger

Un déclencheur spécifie que la base de données doit exécuter automatiquement une fonction donnée chaque fois qu'un certain type d'opération est exécuté. Les fonctions déclencheur peuvent être attachées à une table, une vue ou une table distante.

Exercice

Partie 1 :

Exercice 1

Soit le schéma relationnel d’une agence bancaire régionale.

CLIENT (NUMCL, NOM, PRENOM, ADR, CP, VILLE, SALAIRE, CONJOINT)

DETENTEUR (NUMCL, NUMCP)

COMPTE (NUMCP, DATEOUVR, SOLDE)

Attributs soulignés : Clés primaires. Attributs en italiques: Clés étrangères.

NUMCL et CONJOINT sont définis sur le même domaine.

Écrire un trigger en insertion permettant de contrôler les contraintes suivantes :

- le département dans lequel habite le client doit être 01, 07, 26, 38, 42, 69, 73, ou 74 ;
- le nom du conjoint doit être le même que celui du client.

```
CREATE OR REPLACE FUNCTION check_constraints_on_insert()
RETURNS TRIGGER AS
$$
BEGIN
    -- Check the CP constraint
    IF NEW.CP NOT IN ('01', '07', '26', '38', '42', '69', '73', '74') THEN
        RAISE EXCEPTION 'The department where the client lives must be 01, 07, 26, 38, 42, 69, 73, or 74.';
    END IF;

    -- Check the CONJOINT constraint
    IF NEW.CONJOINT IS NOT NULL THEN
```

```
    DECLARE
        conjoint_nom VARCHAR(100);
    BEGIN
        SELECT NOM INTO conjoint_nom FROM CLIENT WHERE NUMCL = NEW.CONJOINT;
        IF conjoint_nom IS NULL OR conjoint_nom != NEW.NOM THEN
            RAISE EXCEPTION 'The name of the spouse must be the same as the client.';
        END IF;
    END;
  END IF;

    -- If no exceptions were raised, the row is OK
  RETURN NEW;
END;
$$
LANGUAGE 'plpgsql';

CREATE TRIGGER check_constraints_trigger
BEFORE INSERT ON CLIENT
FOR EACH ROW EXECUTE PROCEDURE check_constraints_on_insert();
```

Exercice 2

Soit une table quelconque TABL, dont la clé primaire CLENUM est numérique.

Définir un trigger en insertion permettant d’implémenter une numérotation automatique de la clé. Le premier numéro doit être 1.

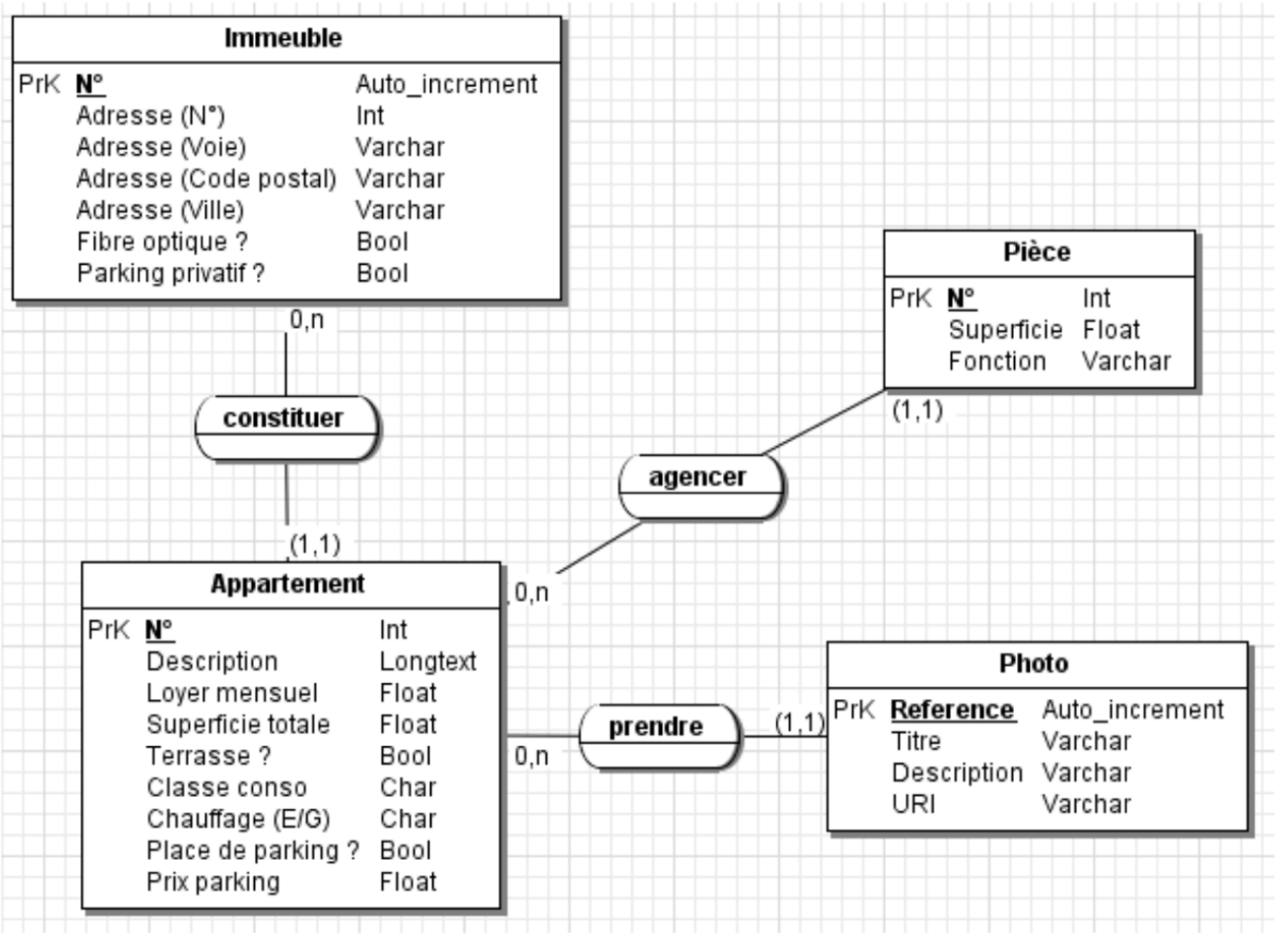
```
CREATE SEQUENCE tabl_seq START 1;

CREATE OR REPLACE FUNCTION increment_key()
RETURNS TRIGGER AS $$
BEGIN
  NEW.clenum = nextval('tabl_seq');
  RETURN NEW;
END;
$$ LANGUAGE 'plpgsql';
```

```
CREATE TRIGGER increment_key_trigger
BEFORE INSERT ON TABL
FOR EACH ROW EXECUTE PROCEDURE increment_key();
```

Partie 2 :

Exercice 1 : Parc immobilier

Soit le modèle de données suivant :



Immeuble(id, adrNum, adrVoie, adrCodePostal, adrVille, fibreOptique, parkingPrivatif) Clef primaire : id

Appartement(#immeuble, num, description, loyer, superficie, terrasse, classeConso, chauffage, placeParking, prixParking)
Clef primaire : immeuble, num
Clef étrangère : immeuble en référence à Immeuble(id)

Piece(#(immeuble, appartement), num, superficie, fonction)
Clef primaire : immeuble, appartement, num
Clefs étrangères : (immeuble, appartement) en référence à Appartement(immeuble, num)

Photo(num, titre, description, uri, #(immeuble, appartement))
Clef primaire : num
Clef étrangère : (immeuble, appartement) en référence à Appartement(immeuble, num)

En voici le script de création de tables :

```
CREATE TABLE Immeuble(
Id INT(11) NOT NULL AUTO_INCREMENT PRIMARY KEY,
adrNum VARCHAR(7) NOT NULL, adrVoie VARCHAR(100) NOT NULL, adrCodePostal VARCHAR(5) NOT NULL,
adrVille VARCHAR(30) NOT NULL, fibreOptique TINYINT NOT NULL, parkingPrivatif TINYINT NOT NULL );

CREATE TABLE Appartement(
immeuble INT(11), num INT(3) NOT NULL, description LONGTEXT,
loyer DOUBLE NOT NULL, superficie DOUBLE NOT NULL,
terrasse TINYINT(1) NOT NULL, classeConso CHAR(1) NOT NULL, chauffage CHAR(1) NOT NULL, placeParking
TINYINT(1) NOT NULL, prixParking DOUBLE,
CONSTRAINT pk_appartement PRIMARY KEY (immeuble, num),
CONSTRAINT fk_immeuble FOREIGN KEY (immeuble) REFERENCES Immeuble(id) );

CREATE TABLE Photo(
immeuble INT(11), appartement INT(3), reference INT(11) NOT NULL,
titre VARCHAR(75), description VARCHAR(255), uri VARCHAR(120) NOT NULL, CONSTRAINT pk_photo

PRIMARY KEY (immeuble, appartement, reference), CONSTRAINT fk_appartement_photo FOREIGN KEY
(immeuble, appartement) REFERENCES Appartement(immeuble, num) );

CREATE TABLE Piece(
immeuble INT(11), appartement INT(3), num INT(2) NOT NULL, superficie DOUBLE ,
fonction VARCHAR(30),
CONSTRAINT pk_piece
```

```
PRIMARY KEY (immeuble, appartement, num), CONSTRAINT fk_appartement_piece FOREIGN KEY (immeuble,
appartement) REFERENCES Appartement(immeuble, num) );
```

Postgresql :

```
-- Création de la table Immeuble
CREATE TABLE Immeuble (
    id SERIAL PRIMARY KEY,
    adrNum VARCHAR(7) NOT NULL,
    adrVoie VARCHAR(100) NOT NULL,
    adrCodePostal VARCHAR(5) NOT NULL,
    adrVille VARCHAR(30) NOT NULL,
    fibreOptique BOOLEAN NOT NULL,
    parkingPrivatif BOOLEAN NOT NULL
);

-- Création de la table Appartement
CREATE TABLE Appartement (
    immeuble INT NOT NULL,
    num INT NOT NULL,
    description TEXT,
    loyer NUMERIC NOT NULL,
    superficie NUMERIC NOT NULL,
    terrasse BOOLEAN NOT NULL,
    classeConso CHAR(1) NOT NULL,
    chauffage CHAR(1) NOT NULL,
    placeParking BOOLEAN NOT NULL,
    prixParking NUMERIC,
    PRIMARY KEY (immeuble, num),
    FOREIGN KEY (immeuble) REFERENCES Immeuble(id)
);

-- Création de la table Photo
CREATE TABLE Photo (
    immeuble INT NOT NULL,
    appartement INT NOT NULL,
    reference SERIAL PRIMARY KEY,
    titre VARCHAR(75),
```

```
    description VARCHAR(255),
    uri VARCHAR(120) NOT NULL,
    FOREIGN KEY (immeuble, appartement) REFERENCES Appartement(immeuble, num)
);

-- Création de la table Piece
CREATE TABLE Piece (
    immeuble INT NOT NULL,
    appartement INT NOT NULL,
    num INT NOT NULL,
    superficie NUMERIC,
    fonction VARCHAR(30),
    PRIMARY KEY (immeuble, appartement, num),
    FOREIGN KEY (immeuble, appartement) REFERENCES Appartement(immeuble, num)
);
```

Questions :

1. Rédiger le trigger permettant de vérifier la contrainte suivante : le prix de la place de parking d'un ap- partement peut et doit être NULL si l'appartement ne possède pas de place de parking. Tester le bon fonctionnement de votre trigger.

```
CREATE OR REPLACE FUNCTION check_parking_price()
RETURNS TRIGGER AS $$
BEGIN
  IF NEW.placeParking = FALSE THEN
    NEW.prixParking = NULL;
  END IF;
  RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER check_parking_price_trigger
BEFORE INSERT OR UPDATE OF placeParking, prixParking ON Appartement
FOR EACH ROW EXECUTE PROCEDURE check_parking_price();
```

2. On souhaite que la contrainte suivante soit vérifiée : la superficie totale d'un appartement doit être égale à la somme de la superficie de chacune de ses pièces. Pour ce faire, créer le trigger qui permet de mettre à jour la superficie d'un appartement à l'insertion d'une pièce.

```
CREATE OR REPLACE FUNCTION update_appartement_area()
RETURNS TRIGGER AS $$
BEGIN
  UPDATE Appartement SET superficie = (SELECT SUM(superficie) FROM Piece WHERE immeuble =
NEW.immeuble AND appartement = NEW.appartement)
  WHERE immeuble = NEW.immeuble AND num = NEW.appartement;
  RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER update_appartement_area_trigger
AFTER INSERT ON Piece
FOR EACH ROW EXECUTE PROCEDURE update_appartement_area();
```

3. Adapter le trigger de la question 1 afin :

- de vérifier la contrainte suivante : un appartement ne peut avoir de place de parking si l'immeuble n'a pas de parking privatif ;
- d'initialiser la superficie de l'appartement à 0 à l'insertion d'un appartement ;
- d'empêcher la modification de la superficie d'un appartement en cas de mise à jour d'un appartement.

```
CREATE OR REPLACE FUNCTION check_appartement()
RETURNS TRIGGER AS $$
DECLARE
immeubleParking BOOLEAN;
BEGIN
   -- a. Un appartement ne peut avoir de place de parking si l'immeuble n'a pas de parking privatif
   SELECT parkingPrivatif INTO immeubleParking FROM Immeuble WHERE id = NEW.immeuble;
   IF immeubleParking = FALSE AND NEW.placeParking = TRUE THEN
      RAISE EXCEPTION 'Un appartement ne peut pas avoir de place de parking si l''immeuble n''a pas
de parking privatif.';
   END IF;

   -- b. Initialiser la superficie de l'appartement à 0 à l'insertion d'un appartement
   IF TG_OP = 'INSERT' THEN
      NEW.superficie = 0;
   END IF;

   -- c. Empêcher la modification de la superficie d'un appartement en cas de mise à jour d'un
```

```
appartement
    IF TG_OP = 'UPDATE' THEN
        IF NEW.superficie != OLD.superficie THEN
            RAISE EXCEPTION 'La superficie d’un appartement ne peut pas être modifiée.';
        END IF;
    END IF;

    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER check_appartement_trigger
BEFORE INSERT OR UPDATE ON Appartement
FOR EACH ROW EXECUTE PROCEDURE check_appartement();
```

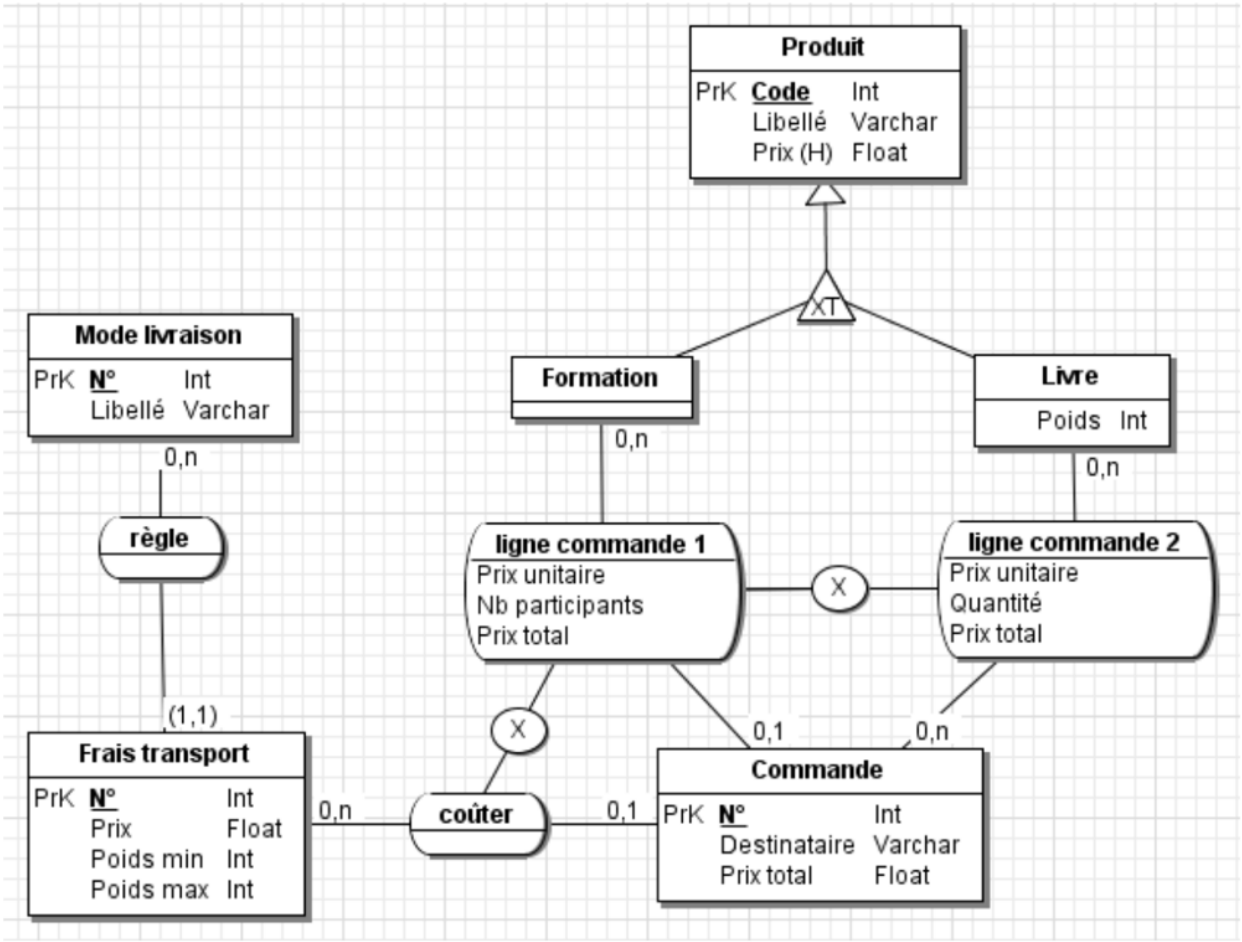
4. En vous inspirant du trigger de la question 2, rédiger celui qui permet de mettre à jour la superficie d’un appartement à la mise à jour d’une pièce. Rédiger également le trigger qui met à jour la superficie d’un appartement à la suppression d’une pièce.

```
CREATE OR REPLACE FUNCTION update_appartement_area_on_update()
RETURNS TRIGGER AS $$
BEGIN
    UPDATE Appartement
    SET superficie = (SELECT SUM(superficie) FROM Piece WHERE immeuble = NEW.immeuble AND
appartement = NEW.appartement)
    WHERE immeuble = NEW.immeuble AND num = NEW.appartement;
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER update_appartement_area_on_update_trigger
AFTER UPDATE OF superficie ON Piece
FOR EACH ROW EXECUTE PROCEDURE update_appartement_area_on_update();

CREATE OR REPLACE FUNCTION update_appartement_area_on_delete()
RETURNS TRIGGER AS $$
BEGIN
    UPDATE Appartement
    SET superficie = (SELECT sum(superficie) FROM Piece WHERE immeuble = OLD.immeuble AND
```

```
appartement = OLD.appartement)
    WHERE immeuble = OLD.immeuble AND num = OLD.appartement;
    RETURN OLD;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER update_appartement_area_on_delete_trigger
AFTER DELETE ON Piece
FOR EACH ROW EXECUTE PROCEDURE update_appartement_area_on_delete();
```

Exercice 2 : Des produits au prix variant au fil du temps

Soit le modèle de données suivant :



Produit(code, libelle, prix, formation, poids)
Clef primaire : code
Commentaire : le champ discrimination formation prend la valeur 1 (vrai) ou 0 (faux)

Mode_Livraison(num, libelle)
Clef primaire : num

Frais_Transport(#livraison, num, prix, poidsMin, poidsMax)
Clef primaire : livraison, num
Clefs étrangères : livraison en référence à Mode_Livraison(num)

Commande(num, destinataire, prixTotal, #(livraion, transport))
Clef primaire : num
Clef étrangère : (livraison, transport) en référence à Frais_Transport(livraison, num)

Ligne_Commande(#commande, #produit, prixu, quantite, prixt) Clef primaire : commande, produit
Clef étrangères :

- commande en référence à Commande(num)
- produit en référence à Produit(code)

Questions :

1. Rédiger le script de création de table relatif à la base de données ci-avant décrite.

```
CREATE TABLE Produit (
    code        SERIAL PRIMARY KEY,
    libelle     VARCHAR(255) NOT NULL,
    prix        DECIMAL(10, 2) NOT NULL
);

CREATE TABLE Formation (
    produit_code    INTEGER PRIMARY KEY REFERENCES Produit(code),
    participants    INTEGER NOT NULL  -- spécifique à Formation
);

CREATE TABLE Livre (
    produit_code    INTEGER PRIMARY KEY REFERENCES Produit(code),
    poids           DECIMAL(10, 2) NOT NULL  -- spécifique à Livre
);
CREATE TABLE Mode_Livraison (
    num         SERIAL PRIMARY KEY,
    libelle     VARCHAR(255) NOT NULL
);
CREATE TABLE Frais_Transport (
    livraison   INTEGER NOT NULL REFERENCES Mode_Livraison(num),
    num         SERIAL NOT NULL,
```

```
    prix        DECIMAL(10, 2) NOT NULL,
    poidsMin    DECIMAL(10, 2) NOT NULL,
    poidsMax    DECIMAL(10, 2) NOT NULL,
    PRIMARY KEY (livraison, num)
);
CREATE TABLE Commande (
    num             SERIAL PRIMARY KEY,
    destinataire    VARCHAR(255) NOT NULL,
    prixTotal       DECIMAL(10, 2) NOT NULL,
    livraison       INTEGER NOT NULL,
    transport       INTEGER NOT NULL,
    FOREIGN KEY (livraison, transport) REFERENCES Frais_Transport(livraison, num)
);
CREATE TABLE Ligne_Commande_Formation (
    commande        INTEGER NOT NULL REFERENCES Commande(num),
    produit         INTEGER NOT NULL REFERENCES Formation(produit_code),
    prixu           DECIMAL(10, 2) NOT NULL,
    participants    INTEGER NOT NULL,  -- Nombre de participants pour la formation
    prixt           DECIMAL(10, 2) NOT NULL,
    PRIMARY KEY (commande, produit)
);
CREATE TABLE Ligne_Commande_Livre (
    commande        INTEGER NOT NULL REFERENCES Commande(num),
    produit         INTEGER NOT NULL REFERENCES Livre(produit_code),
    prixu           DECIMAL(10, 2) NOT NULL,
    quantite        INTEGER NOT NULL,
    prixt           DECIMAL(10, 2) NOT NULL,
    PRIMARY KEY (commande, produit)
);
```

2. Rédiger le trigger permettant de vérifier la contrainte suivante : si une commande porte sur une formation, alors elle ne peut porter sur un ou plusieurs livres.

```
CREATE OR REPLACE FUNCTION check_commande()
RETURNS TRIGGER AS $$
BEGIN
    IF TG_OP = 'INSERT' OR TG_OP = 'UPDATE' THEN
        IF EXISTS (SELECT 1 FROM Ligne_Commande_Formation WHERE commande = NEW.commande)
AND EXISTS (SELECT 1 FROM Ligne_Commande_Livre WHERE commande = NEW.commande) THEN
            RAISE EXCEPTION 'Une commande portant sur une formation ne peut pas comporter un ou
```

```
plusieurs livres.';
        END IF;
    END IF;
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER check_commande_trigger
AFTER INSERT OR UPDATE ON Ligne_Commande_Formation
FOR EACH ROW EXECUTE PROCEDURE check_commande();

CREATE TRIGGER check_commande_trigger
AFTER INSERT OR UPDATE ON Ligne_Commande_Livre
FOR EACH ROW EXECUTE PROCEDURE check_commande();
```

3. Re?diger le trigger permettant de ve?rifier la contrainte re?ciproque : si une commande porte sur un ou plusieurs livres, alors elle ne peut porter sur une formation.

```
CREATE OR REPLACE FUNCTION check_commande_inverse()
RETURNS TRIGGER AS $$
BEGIN
    IF TG_OP = 'INSERT' OR TG_OP = 'UPDATE' THEN
        IF EXISTS (SELECT 1 FROM Ligne_Commande_Livre WHERE commande = NEW.commande) AND
EXISTS (SELECT 1 FROM Ligne_Commande_Formation WHERE commande = NEW.commande) THEN
            RAISE EXCEPTION 'Une commande portant sur un ou plusieurs livres ne peut pas comporter
une formation.';
        END IF;
    END IF;
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER check_commande_inverse_trigger
AFTER INSERT OR UPDATE ON Ligne_Commande_Livre
FOR EACH ROW EXECUTE PROCEDURE check_commande_inverse();

CREATE TRIGGER check_commande_inverse_trigger
AFTER INSERT OR UPDATE ON Ligne_Commande_Formation
FOR EACH ROW EXECUTE PROCEDURE check_commande_inverse();
```

4. Adapter le trigger de la question 2 ou 3 afin de ve?rifier la contrainte suivante : si la commande porte sur une formation, alors il ne doit pas y avoir de frais de transport.

```
CREATE OR REPLACE FUNCTION check_commande()
RETURNS TRIGGER AS $$
BEGIN
   IF TG_OP = 'INSERT' OR TG_OP = 'UPDATE' THEN
      IF EXISTS (SELECT 1 FROM Ligne_Commande_Formation WHERE commande = NEW.num) THEN
         IF EXISTS (SELECT 1 FROM Commande WHERE num = NEW.num AND transport > 0) THEN
            RAISE EXCEPTION 'Une commande portant sur une formation ne doit pas avoir de frais de
transport.';
         END IF;
         IF EXISTS (SELECT 1 FROM Ligne_Commande_Livre WHERE commande = NEW.num) THEN
            RAISE EXCEPTION 'Une commande portant sur une formation ne peut pas comporter un ou
plusieurs livres.';
         END IF;
      END IF;
   END IF;
   RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER check_commande_trigger
AFTER INSERT OR UPDATE ON Commande
FOR EACH ROW EXECUTE PROCEDURE check_commande();

CREATE TRIGGER check_commande_trigger
AFTER INSERT OR UPDATE ON Ligne_Commande_Formation
FOR EACH ROW EXECUTE PROCEDURE check_commande();

CREATE TRIGGER check_commande_trigger
AFTER INSERT OR UPDATE ON Ligne_Commande_Livre
FOR EACH ROW EXECUTE PROCEDURE check_commande();
```

5. Adapter le trigger de la question 2 ou 3 afin que le prix total d’une commande soit automatiquement mis a? jour a? lorsqu’une ligne de commande est inse?re?e.

```
CREATE OR REPLACE FUNCTION update_prix_total_commande()
RETURNS TRIGGER AS $$
BEGIN
```

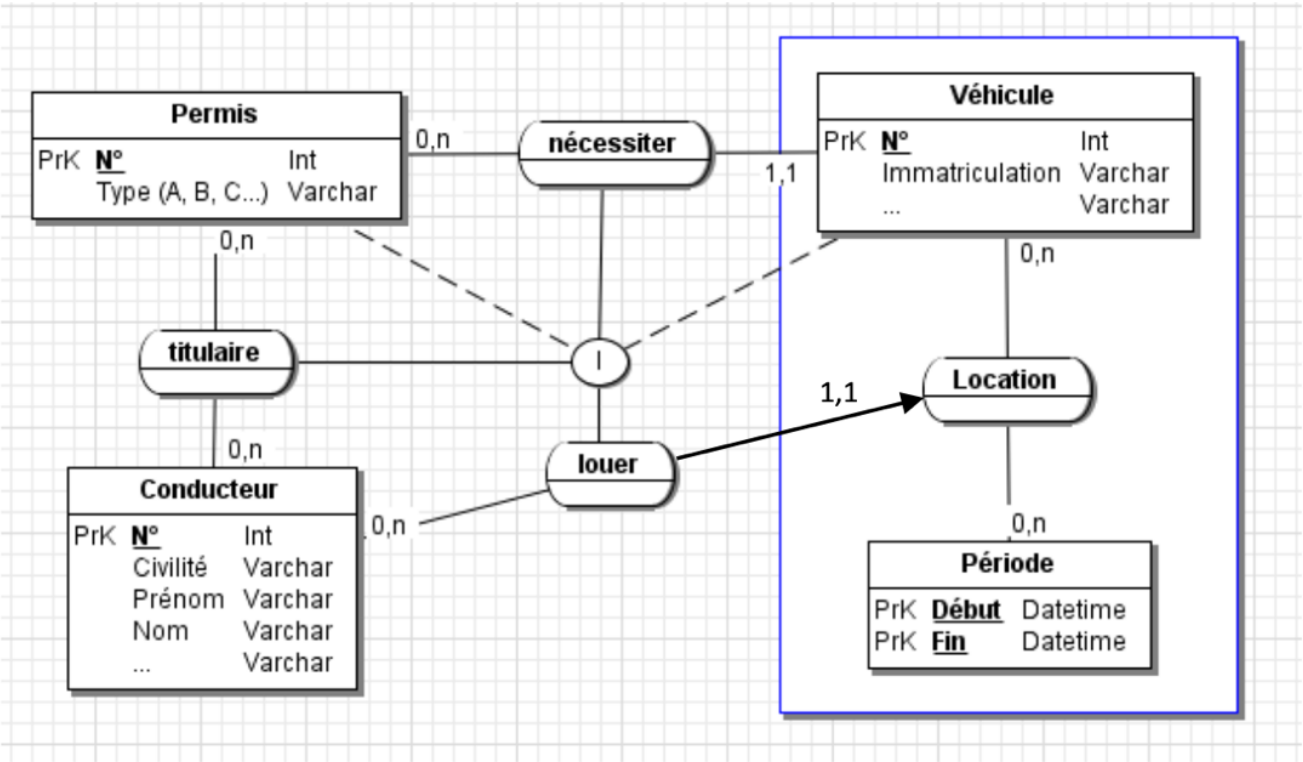
```
    UPDATE Commande
    SET prixTotal = prixTotal + NEW.prixt
    WHERE num = NEW.commande;
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER update_prix_total_commande_trigger
AFTER INSERT ON Ligne_Commande_Formation
FOR EACH ROW EXECUTE PROCEDURE update_prix_total_commande();

CREATE TRIGGER update_prix_total_commande_trigger
AFTER INSERT ON Ligne_Commande_Livre
FOR EACH ROW EXECUTE PROCEDURE update_prix_total_commande();
```

6. Pre?ciser les cas que nous n'avons pas traite?s au travers des questions 1 a? 5.

   - Suppression des lignes de commande : Nous n'avons pas prévu de cas pour la suppression des lignes de commande. Par exemple, si une ligne de commande est supprimée, le prix total de la commande devrait être mis à jour pour refléter ce changement.
   - Modification des lignes de commande : Si une ligne de commande est modifiée (par exemple, changement de la quantité d'un produit), le prix total de la commande devrait également être mis à jour.
   - Modification des frais de livraison : Si le mode de livraison ou les frais de transport associés à une commande sont modifiés, cela peut affecter la validité de la commande (par exemple pour les commandes de formation qui ne doivent pas avoir de frais de transport). Il peut être nécessaire de mettre en place un déclencheur pour gérer cette situation.
   - Rotation des stocks : Notre système actuel ne prend pas en compte la disponibilité des produits. Par exemple, si une commande est passée pour un livre qui n'est plus en stock, nous n'avons aucun moyen de gérer cette situation.
   - Gestion des erreurs et des exceptions : Bien que nous ayons mis en place des déclencheurs pour certaines erreurs, il existe d'autres erreurs potentielles que nous n'avons pas prises en compte. Par exemple, que se passe-t-il si la mise à jour du prix total de la commande échoue ? Ou si un produit est supprimé de la base de données alors qu'il fait encore partie d'une commande ?

Exercice 3 : Location Véhicules

Soit le modèle de données suivant :



Questions :

1. Créer sous MySQL une base de données correspondant au modèle de données ci-dessus (fournir le script de création de tables).

```
CREATE TABLE Permis (
    num         SERIAL PRIMARY KEY,
    type        VARCHAR(10) NOT NULL -- Type de permis (A, B, C...)
);
CREATE TABLE Conducteur (
    num         SERIAL PRIMARY KEY,
    civilite    VARCHAR(10) NOT NULL, -- Civilité (M., Mme., etc.)
    prenom      VARCHAR(255) NOT NULL,
    nom         VARCHAR(255) NOT NULL
);
CREATE TABLE Vehicule (
    num         SERIAL PRIMARY KEY,
    immatriculation VARCHAR(15) NOT NULL,  -- Immatriculation du véhicule
    marque      VARCHAR(255) NOT NULL,    -- Marque du véhicule
    modele      VARCHAR(255) NOT NULL     -- Modèle du véhicule
```

```
);
CREATE TABLE Location (
    num          SERIAL PRIMARY KEY,
    vehicule     INTEGER NOT NULL REFERENCES Vehicule(num),
    conducteur   INTEGER NOT NULL REFERENCES Conducteur(num)
);
CREATE TABLE Periode (
    location     INTEGER NOT NULL REFERENCES Location(num),
    debut        TIMESTAMP NOT NULL,
    fin          TIMESTAMP NOT NULL,
    PRIMARY KEY (location, debut, fin)
);
CREATE TABLE Titulaire (
    conducteur   INTEGER NOT NULL REFERENCES Conducteur(num),
    permis       INTEGER NOT NULL REFERENCES Permis(num),
    PRIMARY KEY (conducteur, permis)
);

CREATE TABLE Necessiter (
    location     INTEGER NOT NULL REFERENCES Location(num),
    permis       INTEGER NOT NULL REFERENCES Permis(num),
    PRIMARY KEY (location, permis)
);
```

2. Rédiger le trigger qui permet, à l'insertion et à la modification d'une Location (INSERT ou UPDATE sur la table « Location »), de vérifier que le conducteur spécifié correspond à un conducteur ayant parmi ses permis celui requis pour conduire le véhicule.

```
CREATE OR REPLACE FUNCTION check_conducteur_permis()
RETURNS TRIGGER AS $$
BEGIN
    IF NOT EXISTS (
        SELECT 1
        FROM Titulaire
        WHERE conducteur = NEW.conducteur
        AND permis IN
            (
            SELECT permis
            FROM Necessiter
            WHERE location = NEW.num
```

```
            )
        ) THEN
            RAISE EXCEPTION 'Le conducteur (%) n''a pas le permis requis pour conduire ce véhicule.',
NEW.conducteur;
    END IF;
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER check_conducteur_permis_trigger
BEFORE INSERT OR UPDATE ON Location
FOR EACH ROW EXECUTE PROCEDURE check_conducteur_permis();
```