8. Créez une procédure qui affiche l'évolution d'un compte bancaire au cours du temps. Le paramètre de la procédure est le numéro du compte bancaire. A chaque fois qu'il y a une opération avec ce compte, une ligne affiche la date de l'opération, avec qui cette opération se fait et quelle est la balance du compte suite à cette dernière.

```
CREATE OR REPLACE FUNCTION preprojet.evolutionCompte(compte CHAR(10)) RETURNS SETOF RECORD AS $$
DECLARE
    solde INTEGER:= 0;
    operation RECORD;
    sortie RECORD;
BEGIN
    FOR operation IN SELECT * FROM exercice.operations WHERE compte_source=compte OR
           compte_destination=compte ORDER BY date_op LOOP
           IF operation.compte_source=compte THEN
                      solde:=solde-operation.montant;
                      SELECT operation.date_op,operation.compte_destination,solde INTO sortie;
                      RETURN NEXT sortie;
           ELSE
                      solde:=solde+operation.montant;
                      SELECT operation.date_op,operation.compte_source,solde INTO sortie;
                      RETURN NEXT sortie;
           END IF;
    END LOOP;
    RETURN;
END;
$$ LANGUAGE 'plpgsql';
SELECT * FROM exercice.evolutionCompte('1234-56789')
    ec(date DATE, solde INTEGER, contrepartie CHARACTER(10));
```

9. Pour chaque compte en banque, ajoutez un champ solde. Ce champ contiendra le solde du compte en banque (somme de tous les montants dont ce compte est destinataire moins la somme de tous les montants dont ce compte est l'origine). Créez un trigger pour mettre ce champ à jour automatiquement.

```
ALTER TABLE preprojet.comptes ADD COLUMN solde
  INTEGER;
```

```
CREATE OR REPLACE FUNCTION preprojet.trigger() RETURNS TRIGGER AS $$
DECLARE
   ancien_solde INTEGER;
BEGIN
   SELECT c.solde FROM preprojet.comptes c
         WHERE c.numero=NEW.compte_source INTO ancien_solde;
   UPDATE preprojet.comptes
         SET solde=ancien_solde-NEW.montant
         WHERE numero=NEW.compte_source;
   SELECT c.solde FROM preprojet.comptes c
         WHERE c.numero=NEW.compte_destination INTO ancien_solde;
   UPDATE preprojet.comptes
         SET solde=ancien_solde+NEW.montant
         WHERE numero=NEW.compte_destination;
   RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER trigger_solde AFTER INSERT ON preprojet.operations
   FOR EACH ROW EXECUTE PROCEDURE preprojet.trigger();
```

GESTION DE LA CONCURRENCE

- **Généralement les BD sont utilisées par plusieurs personnes simultanément.**
  - ex : site web de réservation de place d'avion
- **Risque d'erreurs :**

George	Système	Jerry
Est-ce que la place 3 est libre ?		
	oui	
		Est-ce que la place 3 est libre ?
	oui	
Place George dans 3		
	3 -> George	
		Place Jerry dans 3
	3 -> Jerry	

TRANSACTION

- **Pour gérer la concurrence on utilise le concept de transaction.**
  - Au sein de chaque transaction, tout se déroule comme si on avait l'exclusivité sur la BD
  - Chaque transaction peut soit
    - Réussir (commit) : toutes les modifications de la transaction sont prises en compte, comme si elle avait eu un accès exclusif à la BD.
    - S'annuler (rollback) : toutes les modifications sont annulées, comme si la transaction n'avait jamais eu lieu.

ACID

Acronyme des conditions nécessaires au fonctionnement des transactions

- Atomicité
- Cohérence
- Isolation
- Durable

ATOMICITÉ

Les transactions se terminent soit par

- commit : l'intégralité de la transaction est effectuée
- rollback : l'intégralité de la transaction est annulée, comme si elle n'avait jamais eu lieu

COHÉRENCE

Le contenu de la BD à la fin de la transaction doit être cohérent.

- Pendant la transaction, le contenu peut être incohérent.
- Si le résultat d'une transaction est incohérent, elle est complètement annulée (rollback)

ISOLATION

Si les deux transactions A et B sont exécutées en même temps

- Les modifications de A ne sont pas visibles par B
- Les modifications de B ne sont pas visibles par A

C’est seulement au commit que les modifications deviennent visibles.

DURABLE

Une fois commité, les modifications sont belles et bien présentes dans la BD.

COMPORTEMENT PAR DEFAUT (PUR SQL)

Jusque là nous n'avons jamais géré les transactions !

En dehors de transaction explicite: chaque instruction fonctionne dans sa propre transaction

- commit automatique si l'instruction réussi
- rollback sinon + message d'erreur
- Ceci s'appelle l'auto-commit

BLOC DE TRANSACTION (PUR SQL)

Commence par

```
START TRANSACTION [ mode_transaction]
```

Et se termine par

```
COMMIT
```

Ou bien

```
ROLLBACK
```

EN PL/PGSQL (DONC PAS EN PUR SQL)

- **Chaque fonction est exécutée dans sa propre transaction.**
- **Si il y a une exception non attrapée**
  - Il y a un rollback automatique
- **Comme PostgreSQL ne supporte pas les transactions imbriquées**
  - START TRANSACTION, COMMIT et ROLLBACK sont interdits en PL/pgSQL !

TRANSACTIONS ET TRIGGER

- **Lorsqu'une opération déclenche l'exécution d'un trigger**
  - Ceux-ci sont exécutés dans la même transaction
  - Si la procédure trigger lance une exception : toute la transaction (y compris l'opération) est annulée

- **Possibilité de valider des données**
  - À l'insertion/modification, le trigger valide les données et lance l'exception si elles ne sont pas acceptables

VALIDATION DES DONNEES: EXEMPLE

- **Considérons uniquement la table compte du preprojet**

```
CREATE TABLE comptes (
        numero CHARACTER(10) PRIMARY KEY,
        id_utilisateur INTEGER REFERENCES …
        solde INTEGER NOT NULL DEFAULT 0
);
```

- **Exercice: écrire les opérations déposer/retirer en respectant la contrainte suivante:**
  - les soldes des comptes doivent être positifs

MEILLEURE OPTION ?

```
CREATE TABLE comptes (
numero CHARACTER(10) PRIMARY KEY,
id_utilisateur INTEGER REFERENCES …
solde INTEGER NOT NULL
    CHECK(solde>=0) DEFAULT 0
);


CREATE FUNCTION retirer (compte
CHARACTER(10), montant INTEGER)
RETURNS VOID AS $$
DECLARE
BEGIN
    IF (montant<=0) THEN
        RAISE 'montant negatif';
    END IF;
    UPDATE comptes
    SET solde=solde-montant
    WHERE numero=compte;
    RETURN;
END;
$$ LANGUAGE plpgsql;
```

```
CREATE TABLE comptes (
numero CHARACTER(10) PRIMARY KEY,
id_utilisateur INTEGER REFERENCES …
solde INTEGER NOT NULL DEFAULT 0
);


CREATE FUNCTION retirer (compte
CHARACTER(10), montant INTEGER)
RETURNS VOID AS $$
DECLARE
    s INTEGER;
BEGIN
    IF (montant<=0) THEN
        RAISE 'montant negatif';
    END IF;
    SELECT solde FROM comptes WHERE
numero=compte INTO s;
    IF (s-montant<0) THEN RAISE 'solde
insuffisant');
    END IF;
    UPDATE comptes
    SET solde=solde-montant
    WHERE numero=compte;
    RETURN;
END;
$$ LANGUAGE plpgsql;
```

MAUVAISE SOLUTION: EVITER LES DOUBLES VALIDATIONS

```
CREATE TABLE comptes (
numero CHARACTER(10) PRIMARY KEY,
id_utilisateur INTEGER REFERENCES …
solde INTEGER NOT NULL DEFAULT 0 CHECK (solde>=0) );

CREATE FUNCTION retirer (compte CHARACTER(10), montant INTEGER)
   RETURNS VOID AS $$
DECLARE
   s INTEGER;
BEGIN
   IF (montant<=0) THEN RAISE 'montant negatif';
   END IF;

   SELECT solde FROM comptes WHERE numero=compte INTO s;
   IF (s-montant<0) THEN RAISE 'solde insuffisant');
   END IF;

   UPDATE comptes
   SET solde=solde-montant
   WHERE numero=compte;
   RETURN;
END;
```

AUTRE EXEMPLE

- **Considérons les tables comptes et utilisateurs du préprojet qui ne contiennent pas le champ balance_utilisateur**

```
CREATE TABLE utilisateurs (
      id_utilisateur SERIAL PRIMARY KEY,
      nom VARCHAR(100) NOT NULL CHECK (nom<>''),
      prenom VARCHAR(100) NOT NULL CHECK (prenom<>'‘)
);
CREATE TABLE comptes (
      numero CHARACTER(10) PRIMARY KEY
                        CHECK(numero SIMILAR TO '[0-9]{4}-[0-9]{5}’),
      id_utilisateur INTEGER REFERENCES utilisateurs (id_utilisateur),
      solde INTEGER NOT NULL DEFAULT 0);
```

- **Exercice: écrire les procédures déposer/retirer**
  - En respectant la contrainte suivante: *pour chaque utilisateur, la somme des soldes de ses comptes doit être positif*
    - -> contrainte impossible à formuler dans un CREATE TABLE

MEILLEURE SOLUTION?: SOLUTION 1

```
CREATE FUNCTION retirer(compte CHARACTER(10), montant INTEGER) RETURNS VOID
AS $$
DECLARE
    s INTEGER;
BEGIN
    IF (montant<=0) THEN
        RAISE 'montant negatif';
    END IF;
    SELECT sum(solde) FROM comptes WHERE id_utilisateur =
        (SELECT id_utilisateur FROM comptes WHERE numero=compte) INTO s;
    IF (s-montant<0) THEN RAISE 'solde insuffisant';
    END IF;
    UPDATE comptes
    SET solde=solde-montant
    WHERE numero=compte;
    RETURN;
END;
$$ LANGUAGE plpgsql;
```

MEILLEURE SOLUTION ?: SOLUTION 2

```
CREATE FUNCTION retirer(compte CHARACTER(10), montant INTEGER) RETURNS
   VOID AS $$
DECLARE
BEGIN
   IF (montant<=0) THEN RAISE 'montant negatif';END IF;
   UPDATE comptes SET solde=solde-montant WHERE numero=compte;
   RETURN;
END;
$$ LANGUAGE plpgsql;
```

```
CREATE FUNCTION trigger() RETURNS TRIGGER AS $$
DECLARE
   s INTEGER;
BEGIN
   SELECT sum(solde) FROM comptes WHERE id_utilisateur =
    (SELECT id_utilisateur FROM comptes WHERE numero=NEW.numero) INTO s;
   IF (s<0) THEN RAISE 'solde insuffisant'; END IF;
   RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER trigger_retirer AFTER UPDATE ON comptes
   FOR EACH ROW EXECUTE PROCEDURE trigger();
```

VALIDATION: BONNES PRATIQUES

- **Validation**
  - Au niveau des CREATE TABLE si possible
  - Sinon par trigger
  - Sinon par procédure stockée
  - Rien au niveau de Java
- **Jamais de double validation**