M1–Master Informatique – 2020/2021

Bases de données relationnelles

mars 2021

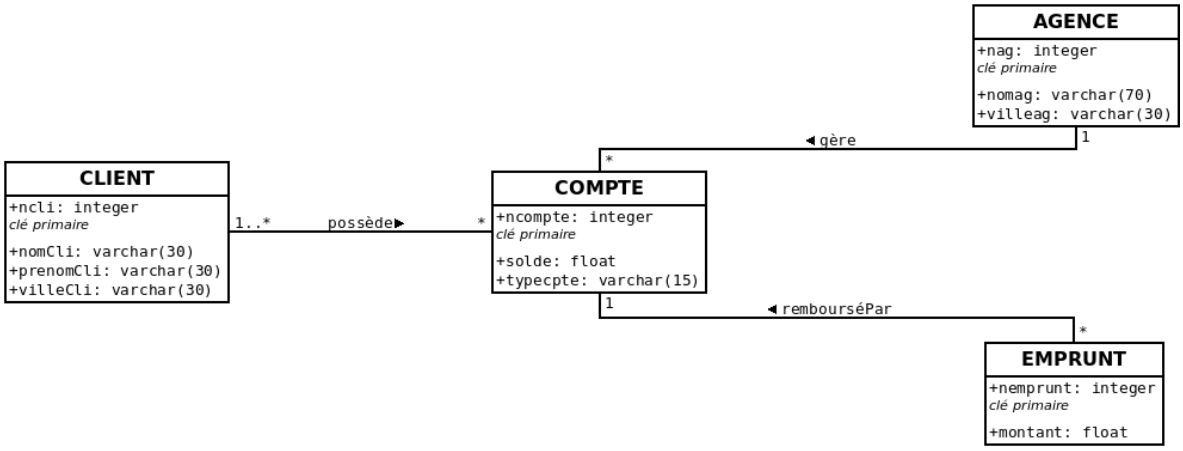
Procédures stockées

Nous allons utiliser PostgreSQL via `psql`, client qui se lance à partir d'un terminal. Pour vous connecter à votre compte, il faut d'abord définir la variable d'environnement `PGHOST` de la façon suivante (n'oubliez pas d'activer le VPN) :

```
export PGHOST=webtp.fil.univ-lille1.fr
```

Exécutez ensuite la commande `psql` dans un terminal pour vous connecter à votre base PostgreSQL : `psql -U <nom_login>`

Exercice 1 :
Dans cet exercice, il s'agit de gérer des emprunts que des clients font auprès d'une banque. Chaque client possède un ou plusieurs comptes, chaque compte étant géré par une agence bancaire. Il y a différents types de compte : des comptes courants, des livrets A, assurances vie, etc. Les comptes peuvent avoir plusieurs propriétaires. Les propriétaires d'un compte peuvent emprunter de l'argent à la banque. Pour rembourser cet emprunt, la banque prélève de l'argent sur ce compte. Le modèle conceptuel de cette base est le suivant :



Voici ci-dessous un schéma relationnel qui traduit ce schéma UML :

```
create table agence(
  nag integer constraint agence_pkey primary key,
  nomag varchar(70) not null,
  villeag varchar(30)
);
create table client(
  ncli integer constraint client_pkey primary key,
  nomcli varchar(30) not null,
  prenomcli varchar(30) not null,
  villecli varchar(30)
);
```

```
create table compte(
  ncompte integer constraint compte_pkey primary key,
  nag integer references agence,
  solde float default 0.0 not null,
  typecpte varchar(15)
);
create table compte_client(
  ncompte integer references compte,
  ncli integer references client,
  constraint compte_client_pkey primary key(ncompte, ncli)
);
create table emprunt(
  nemprunt integer constraint emprunt_pkey primary key,
  ncompte integer not null references compte,
  montant float not null
);
```

Inspectez le fichier banque.sql et exécutez le sur votre base avec la commande suivante : `\i <chemin_vers_le_fichier>`

Question 1.1 : *Affichage des emprunts*
Écrire une fonction stockée `empruntsClient` qui prend en paramètre l'identifiant d'un client et le nom et affiche le prénom de ce client ainsi que les emprunts qu'il a déjà contractés. Cette fonction renvoie `true` quand le client existe et `faux` sinon.
Voici quelques exemples d'exécution :

```
=> select empruntsclient(12);
NOTICE:  client Tsuno Yoko
NOTICE:  emprunt numero 1 de 1000 € sur le compte 978 de l'agence 2
NOTICE:  emprunt numero 3 de 2000 € sur le compte 978 de l'agence 2
NOTICE:  emprunt numero 4 de 4200 € sur le compte 302 de l'agence 1
empruntsclient
----------------
t

=> select empruntsclient(14);
NOTICE:  client Ackerman Mikasa
NOTICE:  pas d'emprunt
empruntsclient
----------------
t

=> select empruntsclient(1);
empruntsclient
----------------
f
```

Les triggers servent très souvent à vérifier des contraintes qui n'ont pas pu être définies lors de la création des tables. C'est le cas dans les questions suivantes.

Question 1.2 : Certains types de compte ne peuvent pas être des comptes joints, c'est-à-dire des comptes possédés par plusieurs clients. C'est le cas des assurances vie ou des livrets A. Complétez le schéma de la base de données à l'aide d'une fonction stockée et du trigger associé afin de faire en sorte que seuls les comptes courants puissent avoir plusieurs propriétaires.
Voici quelques exemples d'exécution :

```
-- le compte 302 appartient déjà au client 12
=> insert into compte_client(ncompte,ncli) values (302,13);
NOTICE:  type de ce compte : cpte courant
INSERT 0 1

-- le compte 529 appartient déjà au client 14 :
=> insert into compte_client(ncompte,ncli) values (529,13);
NOTICE:  type de ce compte : Ass. Vie
ERREUR:  ce compte ne peut être partagé
--> l'insertion est refusée

-- on modifie compte_client : le client 14 possède le compte 145 au lieu du compte 529
-- ce compte 145 appartient déjà au client 10
=> update compte_client set ncompte=145 where ncompte=529 and ncli=14;
NOTICE:  type de ce compte : cpte courant
UPDATE 1

-- on modifie à nouveau compte_client : le client 14 possède le compte 176 au lieu du compte 145
-- ce compte 176 appartient déjà au client 10
=> update compte_client set ncompte=176 where ncompte=145 and ncli=14;
NOTICE:  type de ce compte : livret A
ERREUR:  ce compte ne peut être partagé
--> la modification est refusée
```

Question 1.3 : *Vérification de l'endettement*
Afin d'empêcher le surendettement, la banque a mis en place une nouvelle politique : la somme des montants des prêts liés à un compte doit être inférieure à trois fois le solde de ce compte. Complétez le schéma de la base de données à l'aide d'une fonction stockée et du trigger associé afin d'effectuer automatiquement ce contrôle.
Voici des exemples d'exécution :

```
-- le solde du compte 978 est de 1500€, il est déjà lié à 3000€ d'emprunts
=> insert into emprunt(nemprunt, ncompte, montant) values (11, 978, 1000);
NOTICE:  montant total avant cet emprunt : 3000 ; et après : 4000
INSERT 0 1

=> insert into emprunt(nemprunt, ncompte, montant) values (12, 978, 1000);
NOTICE:  montant total avant cet emprunt : 4000 ; et après : 5000
ERREUR:  limite d'emprunt atteinte
--> l'insertion est refusée
```

Comme le montrent les questions qui suivent, les cardinalités strictement positives peuvent poser des problèmes d'implémentation.
Question 1.4 : Dans le diagramme UML, un compte est toujours géré par 1 agence. Est-ce que le schéma relationnel traduit correctement cette cardinalité ? Comment corriger le problème ?

Question 1.5 : Même question avec la contrainte, "un compte a toujours au moins 1 propriétaire".