Cours SQL

Base de données servant pour le cours

```
drop database if exists cfa;
create database cfa;
use cfa;
create table classe(
    idclasse int (5) not null auto_increment,
    nom varchar(50),
    salle varchar(50),
    primary key(idclasse)
);

insert into classe values (null,"promo 200","salle 5"),(null,"promo 201","salle 4");

create table etudiant(
    idetudiant int (5) not null auto_increment,
    nom varchar (50),
    prenom varchar (50),
    adresse varchar (100),
    idclasse int (5) not null,
    primary key (idetudiant),
    foreign key(idclasse) references classe (idclasse)
);

insert into etudiant values (null,"Raph","lukman","rue de paris",1), (null,"Bait","Anthony","rue de lyon",1),(null,"Sandri

create table professeur(
    idprofesseur int (5) not null auto_increment,
    nom varchar(50),
    prenom varchar (50),
    primary key (idprofesseur)
);

insert into professeur values (null,"Ben","Oka"), (null,"Fred","Launay");

create table enseigner(
    idprofesseur int(5) not null,
    idclasse int(5) not null,
    matiere varchar(50),
    nbheures int(5),
    primary key (idprofesseur,idclasse),
    foreign key (idprofesseur) references professeur (idprofesseur),
    foreign key (idclasse) references classe (idclasse)
);

insert into enseigner values (1,1,"info",8), (1,2,"info",2), (2,1,"anglais",6);
```

Les Vues

Une vue est le résultat d'une requête de selection qu'on peut stocker sous forme d'une table non physique. La vue permet de réaliser des requêtes imbriquées.

Syntaxe :

```
create view nom_vue as (la requete de selection);
drop view nom_vue;
```

Exemple :

Créer une vue qui donne l'état suivant : nom etudiant, prenom etudiant, nom classe, salle de classe.

Solution :

```
create view liste_etu as (
  select e.nom as nom_etudiant, e.prenom as prenom_etudiant, c.nom as nom classe, c.salle
  from etudiant e, classe c
  where e.idclasse = c.idclasse
);
```

On peut executer les requêtes de séléction sur les vues :

```
select * from liste_etu;
```

Afficher le nombre d'étudiants par classe : requête sur la vue

```
select nom_classe,
count(nom_classe) as nb_etudiants
from liste_etu
group by nom_classe;
```

Exercices

Réaliser les vues suivantes :

1. Afficher dans une vue : nom classe, salle, nom prof, prenom prof, matiere, nbheures
   - calculer le nombre de profs par classe
   - afficher le nombre d'heures de cours que dispose chaque classe
   - afficher le nombre d'heures total que enseigne chaque prof
   - Afficher le nombre de classes par professeur

Création de la vue :

```
create view liste_prof as (
  select c.nom as nom_classe, c.salle, p.nom as nom_prof, p.prenom as prenom_prof, e.matiere, e.nbheures
  from classe c, professeur p, enseigner e
  where c.idclasse = e.idclasse and e.idprofesseur = p.idprofesseur
);
```

Requête 1 :

```
select nom_classe,
count(nom_prof) as nb_prof
from liste_prof
group by nom_classe;
```

Requête 2 :

```
select nom_classe, sum(nbheures) as total_heure
from liste_prof
group by nom_classe;
```

Requête 3 :

```
select nom_prof, prenom_prof,
sum(nbheures)
from liste_prof
group by (nom_prof);
```

Requête 4 :

```
select nom_prof, prenom_prof,
count(nom_classe) as nb_classe
from liste_prof
group by (nom_prof);
```

Les Triggers

> Définition : un Triggers ou un declencheur est un ensemble d'ordres SQL qui s'execute automatiquement avant ou apres l'une des requetes suivantes : **insert**, **update**, **delete** sur **une seule table**.

les triggers sont liés à une seule table, donc en théorie, on peut disposer de six **triggers** pour **une seule table** :

Moment : | Ordre : |
Avant **before** | **insert**, **delete**, **update** |
après **after** | **insert**, **update**, **delete** |

Un **trigger** est défini par un **nom**, un **moment** et la **table** sur laquelle il s'**execute**.

Quelle est l'utilité ?

- Vérification de l'intégralité des données :
    - Par exemple avant chaque insertion, les données peuvent être controlées et éventuellement remplacées certaines par des valeurs par défault.
- Historisation des données :
    - Pour ne pas perdre les données définitivement, on peut stocker les données supprimées dans des tables d'archives
- Réaliser des ordres annexes :
    - par exemple, à chaque insertion dans une table, on modifie les attributs d'une autre table à travers des ordres SQL annexes.

Syntaxe de déclaration :

```
create trigger nom_trigger moment ordre_SQL
  on nom_table
  for each row
  begin
```

```
    les ordres SQL à executer
  end;
```

moment : **before**, **after**
ordre_SQL : **insert**, **delete**, **update**
nom_table : table sur laquelle s'execute le trigger

suppression d'un trigger :

```
drop trigger nom_trigger;
```

le caractère delimiteur en SQL est : `;`
Pour executer des ordres SQL dans le triggers avece des points-virgules, il faut changer le délimiteur avant le trigger et le remettre à un point-vigule apreès le trigger. On peut choisir n'import quel caractère : ($, …)

Syntaxe :

```
delimiter $
create trigger ...
  begin
    ordre_SQL;
    ordre_SQL;
    ...
  end $
delimiter;
```

Les triggers s'executent automatiquement sur la table. Il n'y a aucun appel à faire.

Pour manipuler l'enregistrement courant pendant l'execution du trigger, on utiliser deux références : **old** et **new**.
old : ancien enregistrement ou tuple, ou ligne
new : nouvel enregistrement
le tableau suivant décrit l'utilisation de old et new :
insert : **new**
delete : **old**
update : **old**, **new**

Exemple :

1. Ajouter dans la table classe de la base de données CFA :
    - Attribut nb_etudiants
    - l'init a 0
    - supprimer tous les étudiants de la table étudiants
2. Creer un trigger qui oermet a chaque insertion dans la table étudiant l'incrementation de l'attribut nb_etudiants.
3. Creer un trigger qui permet à chaque suppression dans la table étudiant la décrementation de l'attribut nb_etudiants.

Solutions :

1.

```
alter table classe add nb_etudiants int(5);
update classe set nb_etudiants = 0;
delete from etudiant;
```

2.

```
delimiter $
create trigger incrementer after insert
on etudiant
for each row
begin
  update classe set nb_etudiants = nb_etudiants + 1
  where idclasse = new.idclasse;
end $
delimiter ;

insert into etudiant values (null,"Raph","lukman","rue de paris",1);
```

3.

```
delimiter $
create trigger decrementer before delete
on etudiant
for each row
begin
  update classe set nb_etudiants = nb_etudiants  - 1
  where idclasse = old.idclasse;
end $
delimiter ;

delete from etudiant where idetudiant = 4;
```

Les alternatives

Syntaxe :

```
if condition
  then instruction
end if;

if condition
  then instruction;
  else instruction;
end if;
```

Les opérateurs logiques :

and (et), **or** (ou), **not** (non)
is null: tester si l'attribut est **null**

Opérateurs de test :

>, **>=**, **<**, **<=< strong>, =, !=**

Pour déclarer des variables dans un trigger et les utiliser comme dans les programmes :
syntaxe :

```
declare nomvariable type;
```

affectation d'une valeur à une variable :

```
set nomvariable = valeur;
```

on peut insérer un message d'erreur qui affiche en cas d'erreur d'un ordre SQL :

```
insert into Erreur(erreur) value ("message d'erreur");
```

Exercices :

1. creer un trigger qui a chaque insertion d'un etudiant verifie si sa classe a moins de 20 etudiants. si c'est le cas, l'insertion se fera sinon on affiche une erreur.

```
delimiter $
create trigger limiter before insert
  on etudiant
  for each row
  begin
  declare nbetudiants int (5);
  select count(*) into nbetudiants
  from classe where idclasse = new.idclasse;
  if nbetudiants > 20
    then insert into Erreur(erreur) value ("Erreur d'insertion : Il y a trop d'étudiants !");
  end if;
end $
delimiter ;

create table erreur (
  iderreur int(5) not null auto_increment,
  erreur varchar(255),
  primary key (iderreur)
);

update classe set nb_etudiants = 22 where idclasse=1;
insert into etudiant values (null,"Raph","lukman","rue de paris",1);
```

implémentation avec l'héritage :

création de la db :

```
drop database if exists insta;
create database insta;
use insta;

create table etudiant(
    idetudiant int (5) not null auto_increment,
    nom varchar (50),
    prenom varchar (50),
    primary key (idetudiant)
);
create table initial (
  nom varchar(50),
  prenom varchar(50),
  montant float(5.2),
  garant varchar(255),
  idetudiant int(5) not null,
  primary key (idetudiant),
  foreign key (idetudiant) references etudiant(idetudiant)
```

```
);

create table alternant (
  nom varchar(50),
  prenom varchar(50),
  entreprise varchar(200),
  idetudiant int(5) not null,
  primary key (idetudiant),
  foreign key (idetudiant) references etudiant(idetudiant)
);
```

Exercices sur la DB :

Pour la table alternant :

```
delimiter $
create trigger avant_insert before insert
on alternant
for each row
begin
  declare id int(5);
  insert into etudiant values (null, new.nom, new.prenom);
  select idetudiant into id from etudiant where nom = new.nom and prenom = new.prenom;
  set new.idetudiant = id;
end $
delimiter ;

insert into alternant value ("lavigne", "francois", "circus", 1);
```

Pour la table initiale :

```
delimiter $
create trigger avant_insert before insert
on initial
for each row
begin
  declare id int(5);
  insert into etudiant values (null, new.nom, new.prenom);
  select idetudiant into id from etudiant where nom = new.nom and prenom = new.prenom;
  set new.idetudiant = id;
end $
delimiter ;
```

A chaque etudiant supprimé, nous gardons ses informations dans un table archive.
Créer une table Ancien_Etudiants avec les champs suivants :
nom, prenom, date_deepart, status (alternant, initial)

créer un trigger dans la table étudiant qui réalise la suppression et l'archivage.

```
create table Ancien_Etudiants (
  idancient int(5) not null auto_increment,
  nom varchar(50),
  prenom varchar (50),
  date_depart date,
  status enum("alternant", "initial"),
  primary key (idancient)
);

delimiter $
create trigger avant_suppression before delete
on etudiant
for each row
begin
```

```
  declare cpt int(5);
  select count(*) into cpt from alternant
  where idetudiant = old.idetudiant;

  if cpt = 0
    then
      insert into Ancien_Etudiants values (null, old.nom, old.prenom, now(), "initial");
        delete from initial where idetudiant = old.idetudiant;
    else
      insert into Ancien_Etudiants values (null, old.nom, old.prenom, now(), 'alternant');
        delete from alternant where idetudiant = old.idetudiant;
  end if;
end $
delimiter ;

delete from etudiant where idetudiant = 1;
```