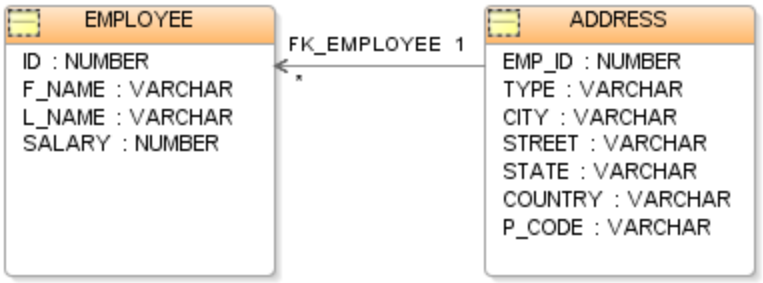
BASES DE DONNEES

Une base de données est un ensemble de données organisées. Un système de gestion de base de données est un logiciel qui va permettre d'interagir avec la base de données pour définir les données que l'on pourra y insérer, créer ou modifier des données, récupérer des données suivants différents critères et administrer la base.

Les bases de données relationnelles sont organisées en un ensemble de relations (ou tables), chacune contenant des données relatives à un même sujet (par exemple une table produits qui contient tous les produits en vente dans un magasin ou une table clients qui contient la liste des clients). Chaque table contient un ensemble d'enregistrements, ou lignes de la table (par exemple dans une table produits, chaque enregistrement contient un produit). Un enregistrement est lui-même composés d'un ensemble de champs de données, ou colonnes ou attributs, qui décrivent l'enregistrement (un client a un nom, un prénom, une adresse, un téléphone, etc.)

Les tables sont souvent connectées entre elles par l'utilisation de clefs étrangères pour indiquer que des données d'une table sont liées à des données d'une autre table. Une table facture contient par exemple un lien vers une table client afin d'assurer que la facture est pour un client référencé, et plusieurs liens vers une table produits pour lister tous les produits facturés.



Afin de pouvoir utiliser une base de données, il est nécessaire de pouvoir définir la structure des données et les liens entre les tables, de pouvoir insérer, modifier ou supprimer des données et enfin de pouvoir interroger les données.

La syntaxe des requêtes SQL se trouve (par exemple) à l'adresse http://www.w3schools.com/sql/ et des rappels sont indiqués plus bas mais de manière brève.

1 Langage de Définition de Données (niveau 1)

Le LDD contient toutes les méthodes pour indiquer la structuration des données. Cela permet notamment de créer des bases de données, des tables dans les bases de données, d'indiquer des contraintes à l'intérieur ou entre les tables, etc.

1.1 Création de bases et de tables

```
CREATE DATABASE <nomTable>;
DROP DATABASE <nomTable>;
```

```
CREATE TABLE <nomtable> (
  <nomAttribut1> <typeAttribut1> <contraintesAttribut1> <valeurParDefautAttribut1>,
  <nomAttribut2> <typeAttribut2> <contraintesAttribut2> <valeurParDefautAttribut2>
);
```

Les attributs peuvent avoir différents types, les plus classiques sont :

Type	Description
CHAR(n)	Chaîne de longueur fixe n
VARCHAR(n)	Chaîne de longueur variable inférieure n
BOOLEAN	Stores TRUE or FALSE values
INTEGER	Nombre entier
DECIMAL(p,s)	Nombre decimal. decimal(5,2) est un nombre avec 3 chiffres avant la virgule, 2 après.
DATE	Stockage de dates

Par exemple si l'on souhaite créer une table pour stocker des commandes : numéro de commande, date de la commande et montant de la commande, on peut faire :

```
CREATE TABLE commandes(
  no_commande CHAR(3),
  date_commande DATE,
  montant NUMERIC(9,2) DEFAULT 0.00
);
```

Exercices (niveau 1) :

Table **CLIENTS**

No_client	Nom	prenom	adresse
105	Dallalon	Jean	5 rue Jean Moulin
101	Rivoire		18 Rue Ronde
102	Favero	André	43 Rue Beaujolais
103	Provent	Catherine	38 Rue du stade
104	Labric		35 Rue des fleurs

Table **PRODUITS**

Reference	Designation	PrixHT
DT010	Disjoncteur 10A	7.21
DT180	Bloc Huger	6.12
DT802	Boîte contrôle	68.35
DT711	Cellule	25.36
DT125	Bloc soc	6.89
DT015	Disjoncteur 15A	14.94
DT205	Brûleur Huger	153.37
DT310	Brûleur soc	200.20
DT120	Connecteur	20.35

sur les **FACTURES**

no_facture	date_facture	Etat
1000	01/01/2009	R
1001	12/02/2009	R
1002	17/03/2009	R
1003	24/04/2009	R
1004	16/05/2009	R
1005	08/07/2009	R
1006	08/07/2009	R
1007	15/07/2009	R
1008	15/07/2009	R
1009	22/07/2009	C
1010	22/07/2009	C
1011	29/07/2009	C
1012	30/08/2009	R
1013	19/10/2009	R

R= réglée C=en cours

sur les **REMPLACEMENTS** (pièces remplacées).

reference	No_interv	Qte_remplacee
DT802	1043	1
DT711	1043	2
DT180	1043	1
DT205	1044	1
DT125	1045	2
DT010	1045	1
DT310	1046	1
DT711	1047	3
DT120	1047	2
DT015	1048	1
DT180	1049	4
DT711	1049	2
DT205	1050	1
DT711	1051	2
DT120	1051	1
DT120	1052	3

sur les INTERVENTIONS

no_interv	Date_interv	Nom_resp	Nom_interv	Temps	No_client	no_facture
1039	10/02/2009	Mauras	Saultier	1	101	1001
1040	10/03/2009	Foucher	Saultier	1	103	1002
1041	15/03/2009	Foucher	Saultier	2	103	1002
1042	06/04/2009	Foucher	Saultier	1	101	1003
1043	03/07/2009	Mauras	Saultier	2	105	1005
1044	04/07/2009	Mauras	Saultier	0.5	101	1006
1045	08/07/2009	Mauras	Bonnaz	1.5	102	1007
1046	10/07/2009	Foucher	Crespin	1	102	1007
1047	11/07/2009	Mauras	Crespin	2	103	1008
1048	15/07/2009	Foucher	Bonnaz	1	105	1009
1049	18/07/2009	Foucher	Saultier	1.5	101	1010
1050	22/07/2009	Foucher	Saultier	0.5	104	1011
1051	23/07/2009	Mauras	Bonnaz	2.5	104	1011
1052	29/07/2009	Mauras	Saultier	1.5	104	1011

Exercice 1. Pour chacune des tables indiquez les types de chacun des champs en justifiant si nécessaire le choix fait quand il y a plusieurs possibilités.

Exercice 2. Ecrivez en SQL la déclaration complète des 5 tables.

1.2 Contraintes sur les attributs

Les contraintes principales qui peuvent s'appliquer à un attribut sont :

- Existence : la valeur de l'attribut ne peut être nulle
- Unicité : deux enregistrements différents ne peuvent avoir la même valeur pour cet attribut
- Clé primaire : ensemble d'attribut qui vérifient : non nuls, uniques et minimal (on ne peut pas enlever d'attribut tout en respectant les contraintes). Dit autrement une clef primaire

identifie chaque enregistrement d'une table de manière unique. On souligne généralement la clef primaire d'une table.

- Clé étrangère : identifie des liens entre plusieurs tables. Une clef étrangère doit être unique dans la table référencée. Par exemple si une table commande contient un numéro de client venant d'une table clients, ce numéro doit permettre d'identifier le client de manière unique. On met un # à côté d'une clef étrangère dans la table qui fait référence (table commande dans l'exemple précédent).
- Contraintes de domaine : indiquent les valeurs possibles pour l'attribut, par exemple l'attribut doit appartenir à une liste de valeurs possibles (H ou F par exemple), à un intervalle de valeurs (entre 0 et 100 pour un pourcentage de remise), être dans un format spécifique (5 chiffres pour un code postal).

```
-- Le montant doit obligatoirement être indiqué
montant NUMERIC(9,2) NOT NULL DEFAULT 0.00
-- Date de commande unique = il ne peut pas y avoir deux commandes le même jour
date_commande DATE UNIQUE
-- Le numéro de commande est la clef primaire de la table = il permet d'identifier
chaque enregistrement de manière unique
no_commande CHAR(3) PRIMARY KEY
-- Le champ client_id est une clef étrangère qui référence le champ P_id de la
table Clients
client_id INT FOREIGN KEY REFERENCES Clients(P_Id)
```

Les contraintes de domaine les plus simples sont :

- IN : permet d'indiquer une liste de valeurs
- BETWEEN AND : permet d'indiquer un intervalle de valeurs
- LIKE : permet d'indiquer que le champ doit s'écrire d'une certaine manière (le signe _ permet de remplacer un caractère quelconque, le signe % permet de remplacer un nombre quelconque de caractères)

```
-- la tva peut valoir 19.6 ou 5.5
tva float not null default 19.6 check (tva IN (19.6, 5.5)),
-- la quantité est entre 1 et 100
quantite integer default 1 check ( quantite BETWEEN 1 AND 100)
-- la référence doit faire trois lettres et commencer par un R
reference char(3) primary key check (reference like 'R__')
-- l'intitulé doit commencer par produit suivi de n'importe quoi
intitule varchar(255) check (intitule like 'produit%')
```

Exercices (niveau 1) :

Exercice 3. Ajoutez les contraintes de clés primaires et étrangères à vos créations de tables.

Exercice 4. On souhaite maintenant rajouter des contraintes sur les différents attributs. En utilisant la carte de référence, ajoutez les contraintes suivantes :

- tous les attributs de ces 5 tables sont obligatoires, sauf l'attribut « prenom » de la table CLIENTS ;
- le numéro de client dans la table CLIENTS est supérieur strictement à 100 ;

- la référence dans la table PRODUITS et REMPLACEMENTS est composée de 5 caractères exactement, et les deux premiers caractères sont 'DT' ;
- l'attribut 'état' dans FACTURES est soit 'C' (en cours), soit 'R' (réglée), et 'C' est sa valeur par défaut ;
- le temps d'intervention ne peut être égal à 0 et est <= 8 heures.

Il est également possible de créer de nouveaux types, appelés domaines ou les contraintes s'expriment comme lors de la création de table :

```
CREATE DOMAIN <nomDomaine> as <type>
CHECK ( <contrainte> );
```

Par exemple :

```
CREATE DOMAIN  intituleType as varchar(255)
check ( VALUE LIKE 'produit%')
```

Exercices (niveau 1) :

Exercice 5. Créer un domaine D_JOUEUR qui permettra de stocker l'identifiant d'un joueur de tennis. Il devra être composé d'exactement 4 caractères, dont le premier est 'j' et les autres seront des chiffres entre 0 et 9.

Exercice 6. Créer un domaine appelé D_TENNIS_SCORE permettant de stocker le résultat d'une rencontre de tennis. Un match de tennis entre joueurs masculins peut être joué en 3, 4, ou 5 sets. Pour gagner un set (on parle aussi de manche), il faut remporter 6 jeux et avoir une avance de 2 jeux minimum sur son adversaire (6-4 ou 3-6 par exemple). Si les joueurs sont à 6-6, un septième jeu les départage : c'est le jeu décisif, "tie-break" en anglais. Un joueur peut donc également gagner 7 jeux à 6.

1.3 Modification de table

Il est possible de modifier la structure d'une table après sa création : ajout ou suppression de champs, ajouts de contraintes de clefs, etc.

```
ALTER TABLE <table> ADD <nomAttribut1> <typeAttribut1>
ALTER TABLE <table> DROP COLUMN <nomAttribut1>
ALTER TABLE <table> ALTER COLUMN <nomAttribut1> <typeAttribut1>
ALTER TABLE <table> ADD PRIMARY KEY (<nomAttribut1>, ...)
```

Exercice (niveau 1) :

Exercice 7. On désire ajouter la ville, le code postal et le téléphone pour chaque client. Ajoutez ces trois attributs dans la table CLIENTS, sachant que le numéro de téléphone est composé d'exactement 14 caractères de format XX-XX-XX-XX-XX. Faites-le en utilisant des requêtes de modification de table.

Exercice 8. Rajoutez la contrainte suivante sur la table : aucun des trois attributs supplémentaires ville, code postal ni téléphone ne peuvent être nuls. Que se passe-t-il ?

2 Langage de manipulation de données

Le LMD permet principalement d'insérer, supprimer et modifier des enregistrements.

```
INSERT INTO nomtable(att1, att2, att3) VALUES (valeur1, valeur2,valeur3)
UPDATE nomtable SET att1=valeur1, att2=valeur2, att3=valeur3 WHERE conditions
DELETE FROM nomtable WHERE conditions
```

Par exemple :

```
-- Insérer un produit de référence R01, de désignation trombone…
INSERT INTO produits (ref_produit, designation, prix_unitaire, no_four)
VALUES ('R01', 'trombone',0.01,'10')
-- Modifier la référence et la désignation du produit R01
UPDATE produits
SET ref_produit='R001', designation = 'Trombone'
WHERE ref_produit='R01';
-- Supprimer tous les produits du fournisseur 1
DELETE FROM produits WHERE no_four=1;
```

Exercice (niveau 1) :

Exercice 9. Insérez des données dans les tables précédentes (pas forcément toutes les données).

Exercice 10. Ecrivez plusieurs requêtes pour tester chaque contrainte que vous avez créé précédemment (exercices 3 et 4).

3 Langage d'interrogation de données (niveau 1)

Le LID permet de récupérer des tuples en faisant différents types de requêtes

3.1 Sélection

```
-- récupérer tous les attributs des enregistrements vérifiant le critère
SELECT *
FROM table
WHERE critère de selection
```

Par exemple :

```
-- tous les produits de prix_unitaire >=155
SELECT *
FROM produits
WHERE prix_unitaire >=155;

-- tous les produits de prix unitaire >=155 du fournisseur 1369
SELECT *
FROM produits
```

```
WHERE prix_unitaire >=155 AND no_four = 1369;
```

Dans la clause where on peut écrire « prix_unitaire >=155 » ou « produits.prix_unitaire >=155. C'est équivalent s'il n'y aucun doute que prix_unitaire vient de la table produits. S'il y a une ambiguïté (à partir du 3.3) il faut préciser la table.

On peut rajouter des contraintes supplémentaires :

```
-- trier les résultats par ordre décroissant (DESC) ou croissant (ASC ou rien)
SELECT …
FROM …
ORDER BY <attribute> [DESC];

-- renommer une table pour simplifier l'écriture : on renomme produits en P
SELECT *
FROM produits P
WHERE P.prix unitaire >=155 AND P.no four = 1369;

-- tous les immeubles ayant 3, 5 ou 7 étages
-- on peut utiliser les autres contraintes de définitions dans la clause WHERE
SELECT * FROM IMMEUBLES
WHERE nbetages IN (3,5,7);
```

3.2 Projection

La projection permet de ne récupérer que certains attributs

```
SELECT attribut_1, attribut_2, …, attribut_k
FROM table
```

Par exemple :

```
-- la référence produit et la désignation de tous les produtis
SELECT ref_produit, designation FROM produits
-- toutes les désignations (sans répétition)
SELECT DISTINCT(designation) FROM produits
-- renommer un attribut
SELECT ref_produit as reference, designation FROM produits;
```

On peut bien entendu mélanger sélection et projection

3.3 Produit cartésien

Le produit cartésien permet de combiner deux (ou plus) tables

```
-- obtenir tous les couples d'enregistrements (e1,e2) où e1 vient de table1 et e2
vient de table2
SELECT *
FROM table1, table2;
```

Un produit cartésien simple mélange des enregistrements sans lien entre eux. On rajoute donc généralement une sélection pour limiter les résultats. Un mélange de produit cartésien et de sélection s'appelle une jointure.

```
-- récupérer tous les couples d'enregistrements qui ont le même numéro de
fournisseur
SELECT *
FROM produits, fournisseurs;
WHERE fournisseurs.no_four  = produits.no_four;
```

3.4 Union

Permet d'obtenir l'union des résultats de deux requêtes

```
SELECT ...
UNION
SELECT ...
```

Il faut que les deux requêtes retournent des résultats ayant le même schéma (même nombre d'attributs, dans le même ordre et de mêmes types).

3.5 Différence et intersection

La différence permet de récupérer des enregistrements en excluant ceux d'une autre requête, l'intersection permet de récupérer les résultats similaires entre deux requêtes. Attention la syntaxe peut fortement varier d'un SGBD à un autre.

```
-- Tous les éléments du premier select moins ceux du second
SELECT ...
EXCEPT
SELECT ...
-- Tous les éléments communs entre le premier et le second select
SELECT ...
INTERSECT
SELECT ...
```

3.6 Requêtes imbriquées

On peut utiliser une requête dans une autre requête

```
-- on cherche tous les résultats de la table1 pour lesquels il existe un résultat
dans table2 qui correspond. C'est similaire à une jointure
SELECT <attributs>
FROM <table1>
WHERE  EXISTS (SELECT * FROM <table2> WHERE table1.x=table2.y)
```

Exercices (niveau 1) :

A partir des tables suivantes (uilisées précédemment avec juste la table tauxhoraire en plus), écrire les requêtes SQL pour répondre aux questions :

```
CLIENTS (noclient, nom, prenom, adresse, ville, cp, tel)
PRODUITS (reference, designation, prixht, qtestock, qtesecurite)
FACTURES (nofacture, datefacture, etat)
REMPLACEMENTS (reference#, nointerv#, qteremplacee)
INTERVENTIONS (nointerv, dateinterv, nomresponsable, nominterv, temps, noclient#,
nofacture#, codetaux#)
```

```
TAUXHORAIRE (codetaux, txhoraire)
```

Exercice 11. Référence et désignation des produits dont le prix est supérieur à 50 euros.

Exercice 12. Numéro, date et temps passé des interventions effectuées par Crespin.

Exercice 13. Nom, prénom et numéro de téléphone des clients, triés par ordre alphabétique croissant.

Exercice 14. Numéro et date des factures réglées triées par ordre décroissant.

Exercice 15. Désignation des produits dont la différence entre quantité en stock et quantité de sécurité est comprise dans l'intervalle [1;10].

Exercice 16. Référence et désignation des produits dont la référence se termine par le chiffre 1.

Exercice 17. Référence des produits remplacés au moins 2 fois lors de la même intervention, triées par ordre croissant. Chaque produit ne doit apparaitre qu'une fois.

Exercice 18. Nom du client, nom de l'intervenant et date de l'intervention pour toutes les interventions.

Exercice 19. Numéro et date des interventions relatives à la facture réglée le 15/07/2009.

Exercice 20. Numéro des factures affectées au client *Rivoire*.

Exercice 21. Désignation des produits remplacés lors de l'intervention du 03/07/2009.

Exercice 22. Numéro des factures non réglées, en ordre croissant, et nom du client correspondant.

Exercice 23. Date des factures intégrant le remplacement de *Bruleurs*, et pour lesquelles l'intervenant est *Saultier*. Gérez la casse (minuscules et majuscules).

Exercice 24. Désignations des produits remplacés pour le client *Provent*.

Exercice 25. Numéro des clients chez lesquels Saultier est intervenu mais pas Bonnaz.

Exercice 26. Numéro et date des factures réglées du client Rivoire, avec en plus numéro et date des factures en cours du client Dallalon.

4 LID – agrégation (niveau 2)

L'agrégation permet de faire des calculs sur les résultats au lieu de simplement les retourner.

```
-- Nombre de familles enregistrées dans la base
SELECT COUNT(*) FROM familles ;
-- Nombre de familles enregistrées dans la base pour lesquelles une profession est
déclarée
SELECT COUNT(profession) FROM familles ;
-- Nombre de professions différentes occupées par les familles enregistrées dans la
base
SELECT COUNT(DISTINCT profession) FROM familles ;
```

```
-- nombre de membres des familles les plus nombreuses
SELECT MAX(nbpersonnes) FROM familles;
```

```
-- superficie moyenne des appartements
SELECT AVG(superficie) FROM appartements;
-- nombre total de personnes composant les familles de la base
SELECT SUM(nbpersonnes) FROM familles;
```

La constitution de groupes permet de calculer des résultats par groups et pas sur touet la base. La forme générale est

```
SELECT [DISTINCT] att1, att2, ... att_n, aggregat (expression)
FROM tables
WHERE conditions
GROUP BY att1, att2, ... att_n, …;
```

Par exemple :

```
-- nom et nombre d'appartements en regroupant par nom = nombre d'appartement par
nom
SELECT nom, COUNT(noapp)
FROM familles F, occupants O
WHERE O.idfam=F.idfam
GROUP BY nom ;
```

On peut pas utiliser d'agrégats dans la clause WHERE (comme par exemple WHERE SUM(montant)<100). Pour cela on dispose de HAVING qui fonctionne de la même manière.

```
-- nom et nombre d'appartements en regroupant par nom en se limitant aux familles
possédant plus de 2 appartements.
SELECT nom, COUNT(noapp)
FROM familles F, occupants O
WHERE O.idfam=F.idfam
GROUP BY nom, F.idfam
HAVING count(noapp) >2 ;
```

Exercices (niveau 2) :

Ecrire les requêtes SQL demandées en utilisant la base suivante (il n'est pas nécessaire de créer la base) :

```
Membre (NumBuveur, NomB, PrenomB, VilleB)
Viticulteur (NumVitic, NomV, PrenomV, VilleV)
Vin (NumVin, Cru, Millesime, Region, NumVitic#)
Commande (NumCom, NumBuveur#, NumVin#, QtteCommandee, DateCom)
```

Exercice 27. Liste des régions avec son nom et le nombre de vins produits ;

Exercice 28. Pour chaque membre de Paris, nom, le numéro et la quantité moyenne commandée ;

Exercice 29. Les vins (numéro, cru, nombre de commandes) ayant été commandés au moins deux fois.

Exercice 30. Expliquer en français ce que fait chaque requête ci-dessous :

```
SELECT DISTINCT millesime
FROM Vin
WHERE NOT EXISTS (SELECT * FROM Commande WHERE Vin.NumVin=Commande.NumVin)
ORDER BY millesime;
```

```
SELECT numBuveur, NomB, VilleB
FROM Membre
WHERE VilleB = (SELECT VilleB From Membre WHERE NumBuveur=1400)
EXCEPT
SELECT numBuveur, NomB, VilleB
FROM Membre
WHERE NumBuveur=1400;
```

```
SELECT Region, SUM(QtteCommandee)
FROM VIN
LEFT JOIN Commande ON Vin.NumVIn=Commande.NumVin
GROUP BY Region
HAVING SUM(QtteCommandee)<10 OR SUM(QtteCommandee) IS NULL;
```

```
SELECT NumBuveur
FROM Buveur
WHERE NOT EXISTS (
  SELECT NumVin
  FROM Vin
  WHERE NOT EXISTS (
    SELECT *
    FROM Commande
    WHERE Commande.NumVin=VIn.NumVin
      AND Commande.NumBuveur=Buveur.NumBuveur))
```

5 Règles et triggers (niveau 2 et 3)

Les règles et triggers permettent d'exécuter automatiquement des actions (requêtes SQL ou traitements plus complexes) en réponses à des modifications de la base (évènement déclencheur).

5.1 Règles (niveau 2)

Les règles permettent d'exécuter des procédures simples quand un événement a lieu, on peut faire autre chose à la place (INSTEAD) ou en plus (ALSO) :

```
CREATE RULE <nomregle> AS
```

```
ON <evenement_declencheur> TO <nomtable>
DO  ALSO | INSTEAD    <traitement>
```

Par exemple si tentative d'insertion de l'utilisateur test, rien ne se passe :

```
CREATE RULE refus_insert AS
ON  INSERT TO magasin
WHERE current_user = 'test'
DO INSTEAD NOTHING;
```

Si l'on souhaite exécuter un traitement plus utile, on a toujours accès aux nouveaux enregistrements (lors d'un insert ou update) par la variable NEW, et aux anciens (lors d'un delete ou update) par la variable OLD, par exemple en cas de suppression dans la table produit on stocke le produit dans une table archives :

```
CREATE RULE archivage
AS ON DELETE
TO PRODUITS
DO ALSO (INSERT INTO archives VALUES (OLD.reference, OLD.nomProduit));
```

Exercices (niveau 2)

```
GENRE (code_genre, libelle_genre, nb_livres)
EXEMPLAIRE (no_exemplaire, no_livre#, date_achat, prix_achat)
LIVRE (no_livre, titre livre, nom traducteur, nom auteur, type public, code genre#)
LECTEUR  (no_lecteur,  nom_lecteur,  prenom_lecteur,  rue_lecteur,  ville_lecteur,
cp_lecteur, tel_lecteur, date-naissance)
EMPRUNTER (date_emprunt, no_exemplaire#, no_lecteur#, duree)
STATS ( numero, nom, prenom, nbre_livres, duree_totale, duree_moyenne);
```

Exercice 31. A partir de la base précédente, créez trois règles permettant de mettre à jour le champ nb_livres de la table GENRE quand on ajoute (respectivement, que l'on retire ou que l'on met à jour) un livre dans la table LIVRE.

Exercice 32. La table Stats contient les statistiques des emprunts de chaque lecteur. Quelles règles sont nécessaires pour que cette table soit toujours à jour. Ecrire ces règles.

5.2 Triggers (niveau 3)

Les triggers permettent d'exécuter des traitements plus complexes (for, if, etc.). La syntaxe permet d'exécuter une procédure avant ou après un événement. La procédure peut être exécutée pour chaque ligne (ROW) impactée par l'événement ou pour chaque requête SQL (STATEMENT) :

```
CREATE TRIGGER nomtrigger
{ BEFORE | AFTER} { event [ OR event…]}
ON nomtable
{ FOR EACH ROW | STATEMENT}
        EXECUTE PROCEDURE fonctionDeclencheur ();
```

L'utilisation de triggers nécessite de savoir écrire des procédures SQL. On peut es écrire par exempe avec le langage PLPGSQL (sous POSTGRESQL). Voici un exemple qui insère un

produit dans une table Archives si la désignation avant l'événement (OLD.designation) est différente de la désignation après l'événement (NEW.designation). Si la désignation n'a pas changé on affiche un avertissement (RAISE NOTICE)

```
CREATE FUNCTION Archivage()
RETURNS TRIGGER
AS '
BEGIN
         IF NEW.designation != OLD.designation
         THEN
                  INSERT INTO Archives  VALUES
                  (OLD.ref_produit, current_date,OLD.designation);
         ELSE
                  RAISE NOTICE 'Pas de modification de la designation';
         END IF;
         RETURN NEW;
END;'   LANGUAGE 'plpgsql';
```

On peut également faire un RAISE EXCEPTION 'message' qui interrompt tout le traitement (y compris l'événement déclencheur si le TRIGGER est BEFORE).

Dans une procédure on peut exécuter une requête SQL et récupérer le résultat, par exemple :

```
SELECT INTO nbRes count(*) FROM Produits;
IF nbRes=0 THEN
         RAISE NOTICE 'table vide';
END IF;
```

Exercices (niveau 3)

On considère la table suivante :

```
FILMS ( nofilm, titre, norealisateur#, noproducteur#)
JOUER ( nofilm# , noacteur#)
VOIR (nofilm# , nospect#)
AIMER (nofilm# , nospect#)
INDIVIDUS (nopers , nompers)
```

Exercice 33. Créez un trigger et la procédure associée pour qu'à chaque ajout dans la table aimer un enregistrement soit ajouté dans la table voir (un spectateur ne peut pas aimer un film sans l'avoir vu)

Exercice 34. Créez un trigger et la procédure associée pour qu'à chaque suppression d'un enregistrement dans la table VOIR, l'enregistrement correspondant dans la table AIMER soit supprimé, s'il existe. Quelle solution adoptez-vous (attention : on n'aime pas nécessairement tous les films que l'on voit) ? Ecrivez la solution que vous proposez.

6 Mini-projet (niveaux 1 à 3)

En reprenant tous les points précédents, on souhaite concevoir une base de données complète. La base de données sert à gérer des articles de maroquinerie et des clients. Toutes les informations que vous avez pu récupérer auprès du vendeur sont indiquées plus bas. Vous devez identifier les différentes tables qui seraient utiles, les attributs de ces tables ainsi que les clefs primaires et étrangères. Vous devez ensuite écrire les requêtes SQL pour créer les tables, clefs, insertion de données, règles et triggers si nécessaire, etc.

SACS A MAIN

Référence	Désignation	Matière
1121	Sac Longchamp	Crocodile
1122	Sac Montagne verte	Serpent
1123	Sac Lipsy	Sky
1124	Sac Longchamp	Serpent
...		

CLIENTS

Nom client	Adresse	Catégorie
Dupont	4 rue de Condé 69002 Lyon	Commerçant
Durand	2 rue neuve 69001 Lyon	Particulier
Dubois	8 place de la Bourse 69002 Lyon	Commerçant
Durand	17 rue des Myosotis 74012 Paris	Particulier
...		

Prix de vente d'un sac :

Numéro : 1121

Type de client : commerçant Prix de vente : 293 €

Type de client : particulier Prix de vente : 369 €

Numéro : 1122

Type de client : commerçant Prix de vente : 258 €

Type de client : particulier Prix de vente : 342 €

7 Indexation, optimisation (niveau 3)

Concevoir une base de données ne se limite pas à la structure mais également à faire en sorte que les requêtes s'exécutent efficacement. Cela passe notamment par l'utilisation d'index qui est un outil utilisé par le SGBD essentiellement pour retrouver rapidement les données.

Pour comprendre un peu le fonctionnement des index on peut notamment utiliser la commande EXPLAIN qui permet d'obtenir des informations sur l'exécution de la requête. Par exemple :

```
EXPLAIN SELECT * FROM `adherents` WHERE Nom = "guillaume"
```

id	select_type	table	type	possible_keys	key	key_len	Ref	rows	Extra
1	SIMPLE	adherents	ALL	*NULL*	*NULL*	*NULL*	*NULL*	271561	Using where

Ce qui nous intéresse ici est que le SGBD n'utilise pas d'index et doit donc chercher dans tous les enregistrements (271561). On rajoute maintenant un index sur le champ Nom :

```
CREATE INDEX debutDeNom ON adherents (Nom(10));
EXPLAIN SELECT * FROM `adherents` WHERE Nom = "guillaume"
```

id	select_type	table	type	possible_keys	key	key_len	Ref	rows	Extra
1	SIMPLE	adherents	ref	Nom	Nom	768	Const	89	Using index condition

Ici la recherche est effectuée sur 89 enregistrements et l'index est bien utilisé. On imagine bien le gain de temps en cherchant dans 89 enregistrements plutôt que dans 271561.

Certains SGBD indiquent aussi le temps qu'une requête a pris. Cette information doit être utilisée de manière précautionneuse mais peut tout de même être pertinente. A noter que mis à part sur des très grandes tables ou lors de requêtes impliquant plusieurs tables, les requêtes prennent très peu de temps et il n'est pas évident de juger de la pertinence d'une optimisation.

Exercices (niveau 3) :

Les exercices de cette partie sont moins guidés et doivent vous permettre de comprendre un peu mieux ce que sont les index et comment ils peuvent servir. Vous pouvez dans cette partie utiliser n'importe quelle base de données mais pour que cela soit pertinent il vaut mieux que la base contienne beaucoup d'enregistrements.

Exercice 35. Quels sont les champs utiles lorsque l'on utilise EXPLAIN.

Exercice 36. Quels sont les types d'index les plus courants.

Exercice 37. Quels sont les avantages et les inconvénients à utiliser des index ? Quand faut-il en utiliser ?

Exercice 38. Créez une table et remplissez la avec beaucoup d'enregistrements (on trouve facilement des scripts sur internet pour remplir une base de données, notamment avec des

procédures stockées). Faites des tests de requêtes simples avec ou sans index. En particulier testez des requêtes du type

```
SELECT * FROM maTable WHERE Nom LIKE "toto%";
SELECT * FROM maTable WHERE Nom LIKE "%toto";
```

Exercice 39. Vous pouvez dans le cas d'un champ texte fixer la longueur de l'index. Testez avec différentes longueurs et des requêtes de l'exercice précédent, par exemple :

```
CREATE INDEX debutDeNom ON adherents (Nom(1));
EXPLAIN SELECT * FROM `adherents` WHERE Nom LIKE "guill%"
```

L'optimisation ne passe pas seulement par l'utilisation d'index mais d'autres points sont à prendre en compte :

Exercice 40. Pourquoi faut-il éviter les "SELECT *" en règle générale.

Exercice 41. Qu'est-ce que la dénormalisation dans le cadre de l'optimisation de bases de données. Créez une base pour montrer les avantages que cela peut avoir (par exemple avec une requête sur une seule table opposé à deux tables avec une jointure).

Exercice 42. Renseignez-vous sur les différences entre le row-level lock et le table-level lock.

Exercice 43. Voyez-vous d'autres vecteurs d'optimisation possibles ?

8 Bases NoSQL (niveau 3)

Les bases NoSQL (Not Only SQL qui se prononce No-S-Q-L ou No-Sequel selon) permettent de stocker et d'interroger des données suivants d'autres modalités que la structure relationnelles vue précédemment.

Il n'y a pas de modèle unique mais différents modèles (en particulier clefs-valeurs, colonnes, documents, graphes) selon les besoins que l'on peut rencontrer.

Exercices (niveau 3)

Exercice 44. Renseignez-vous sur les 4 modèles principaux cités précédemment, leurs avantages et inconvénients.

Une base de données intéressante est Neo4j qui est une base graphes très pertinente pour stocker des relations entre objets (par exemple un réseau d'amis sur facebook) et faire des requêtes sur le graphe (qui sont les amis de mes amis par exemple).

Neo4j s'installe très simplement et possède une interface web assez claire et un cours (incluant des interactions avec une base réelle) est disponible à l'adresse suivante : http://neo4j.com/graphacademy/online-course-getting-started/. Il faut s'inscrire mais le mail ne sert à rien sauf à continuer le cours, vous pouvez donc indiquer une fausse adresse (ou une vraie).

Exercice 45. Faites une formation minimum sur Neo4j.

9 Premiers pas sous postgreSQL et pgadmin

a. Lancez pgadmin puis double-cliquez sur le serveur pour vous connecter.

b. Faites un clic droit sur « Bases de données » puis « Ajouter une base de données ». Comme vous allez réutiliser cette base, nommez–la « Gestion ».

c. Cliquez sur votre nouvelle base puis sur l'icône SQL. Dans la fenêtre qui vient de s'ouvrir, vous pouvez taper du code SQL dans la partie supérieure de la fenêtre puis l'exécuter en cliquant sur l'icône correspondant :

   - Tapez « `create table test () ;` » puis exécutez la requête. Vérifiez dans la fenêtre principale de pgadmin en ouvrant « Gestion / Schémas /public / Tables » que la table test a bien été créée.
   - Refaites la même commande, que se passe-t-il ?
   - Pour éviter ce problème, ajoutez « `drop table if exists test ;` » au début de votre script.

d. Effacez tout le script puis créez la table « clients » en SQL. Exécutez le script pour vérifier que la syntaxe est bonne. Que se passe-t-il si vous exécutez le script à nouveau ? Remédiez-y. Sauvegardez votre script en l'appelant gestion.sql.

e. Afin de fixer le style de date au format européen (où le jour est avant le mois) rajoutez la commande suivante au tout début du script :

   « `set datestyle to 'european'` »;

f. Créez toutes les autres tables avec les contraintes définies précédemment.

g. Insérez les enregistrements de la table CLIENTS à l'aide de plusieurs commandes INSERT.

h. Récupérez le fichier scriptgestion.sql mis à votre disposition sur Moodle et contenant les lignes nécessaires au peuplement des 4 autres tables (hormis CLIENTS, que vous venez de peupler à la question 1). Ajoutez le contenu de ce fichier à la suite de votre propre script.