Introduction aux langage SQL

Alexandre Meslé

15 mai 2018

Table des matières

1 Notes de cours **2**

1.1 Introduction . . . 2
1.1.1 Qu'est-ce qu'un SGBDR ? . . . 2
1.1.2 SQL . . . 2
1.1.3 Connexion à une base de données . . . 2
1.1.4 Consultation des tables . . . 3
1.1.5 Organisation relationnelle des données . . . 3

1.2 Contraintes déclaratives . . . 5
1.2.1 Valeurs par défaut . . . 5
1.2.2 Champs non renseignés . . . 5
1.2.3 Clé primaire . . . 5
1.2.4 Clé étrangère . . . 5
1.2.5 Syntaxe alternative . . . 6

1.3 Introduction aux requêtes . . . 7
1.3.1 Compléments sur `SELECT` . . . 7
1.3.2 Instruction `WHERE` . . . 7
1.3.3 Conditions . . . 7
1.3.4 Suppression . . . 9
1.3.5 Mise à jour . . . 9

1.4 Jointures . . . 10
1.4.1 Principe . . . 10
1.4.2 Produit cartésien . . . 10
1.4.3 Jointure . . . 11
1.4.4 Jointures réflexives . . . 11

1.5 Agrégation de données . . . 13
1.5.1 Fonctions d'agrégation . . . 13
1.5.2 Groupage . . . 14

1.6 Vues . . . 16
1.6.1 Définition . . . 16
1.6.2 Syntaxe . . . 16
1.6.3 Application . . . 16
1.6.4 Suppression . . . 17

1.7 Requêtes imbriquées . . . 18
1.7.1 Sous requêtes renvoyant une valeur scalaire . . . 18
1.7.2 Sous requêtes renvoyant une colonne . . . 19
1.7.3 Sous requêtes non correlées renvoyant une table . . . 20
1.7.4 Sous requêtes correlées . . . 21

1.8 Procédures stockées . . . 23
1.8.1 Exemple . . . 23
1.8.2 SQL Procédural . . . 23
1.8.3 Procédures . . . 23
1.8.4 Curseurs . . . 24
1.8.5 Triggers . . . 25

.1 Scripts de création de tables . . . 26
.2 Livraisons Sans contraintes . . . 26
.3 Modules et prerequis . . . 26
.4 Géométrie . . . 28
.5 Livraisons . . . 28
.6 Arbre généalogique . . . 30
.7 Comptes bancaires . . . 30
.8 Comptes bancaires avec exceptions . . . 32
.9 Secrétariat pédagogique . . . 34
.10 Mariages . . . 35
.11 Bibliothèque . . . 36

Chapitre 1

Notes de cours

1.1 Introduction

1.1.1 Qu'est-ce qu'un SGBDR ?

Un **SGBD** (Système de Gestion de Bases de Données) est un logiciel qui stocke des données de façon organisées et cohérentes. Un **SGBDR** (Système de Gestion de Bases de Données Relationnelles) est le type particulier de SGBD qui fera l'objet de ce cours. Il vous sera décrit plus tard ce qui fait qu'une bases de données est relationnelle.

Les bases de données les plus répandues sont :

- **Oracle**, qui est considéré comme un des SGBDR les plus performants.
- **Microsoft SQL Server**, la mouture de microsoft, qui est intégré au framework .NET.
- **mySQL**, un logiciel libre fonctionnant souvent de pair avec Apache et Php, et de ce fait très répandu dans la programmation web.
- **Access**, incorporé à Microsoft Office. Il présente l'énorme avantage de mettre à disposition de l'utilisateur une interface graphique. En contrepartie il est mono-utilisateur et de faible capacité.

Les SGBDRs sont généralement des serveurs auxquels des clients se connectent, il doivent supporter plusieurs connections simultanées. Les clients dialoguent alors avec le serveur pour lire ou écrire des donnés dans la base.

1.1.2 SQL

Le SQL, Structured Query Language, est un langage Standard permettant à un client de communiquer des instructions à la base de données. Il se décline en quatre parties :

- le **DDL** (Data definition language) comporte les instructions qui permettent de définir la façon dont les données sont représentées.
- le **DML** (Data manipulation language) permet d'écrire dans la base et donc de modifier les données.
- le **DQL** (Data query language) est la partie la plus complexe du SQL, elle permet de lire les données dans la base à l'aide de requêtes.
- le **DCL** (Data control language), qui ne sera pas vu dans ce cours permet de gérer les droits d'accès aux données.

A cela s'ajoute des extensions procédurales du SQL (appelé PL/SQL en Oracle). Celui-ci permet d'écrire des scripts exécutés par le serveur de base de données.

1.1.3 Connexion à une base de données

Dans une base de données relationnelle, les données sont stockées dans des tables. Une **table** est un tableau à deux entrées. Nous allons nous connecter à une base de données pour observer les tables.

Oracle

Sous oracle, le client s'appelle SQL+, le compte utilisateur par défaut a pour login `scott` et password `tiger`.

La liste des tables s'affiche en utilisant l'instruction

```
SELECT TABLE_NAME FROM USER_TABLES
```

mySQL

La méthode la plus simple pour s'initier à mysql est d'utiliser un kit de easyphp, wamp, etc. Vous disposez dans ce cas d'une option vous permettant d'ouvrir une console mysql.

La liste des bases de données stockées dans le serveur s'obtient avec l'instruction

```
show databases

Database
information_schema
arbreGenealogique
banque
geometrie
livraisons
playlist
repertoire
secretariat
```

On se connecte à l'une des bases de données avec l'instruction

```
use nomdelabase
```

1.1.4 Consultation des tables

Une fois dans la base, on obtient la liste des tables avec l'instruction

```
show tables

Tables_in_livraisons
DETAILLIVRAISON
FOURNISSEUR
LIVRAISON
NB_FOURNISSEURS_PAR_PRODUIT
NB_PROD_PAR_FOU
PRODUIT
PROPOSER
```

On affiche la liste des colonnes d'une table avec l'instruction

```
desc PRODUIT

Field    Type     Null     Key     Default  Extra
numprod  int(11)  NO       PRI     NULL
nomprod  varchar(30)       YES             NULL
```

Le contenu d'une table s'affiche avec l'instruction

```
SELECT *
FROM PRODUIT

numprod nomprod
1       Bocal de cornichons
2       Tube de dentifrice
3       Flacon de lotion anti-escarres
4       Déodorant fraîcheur 96 heures
```

1.1.5 Organisation relationnelle des données

Nous utiliserons pour commencer les types suivants :

— numérique entier : **int**

— numérique à point fixe : **number** (Oracle seulement)
— numérique à point flottant : `real`
— chaîne de caractères : **varchar**(`taille`) ou **varchar2**(`taille`) (Oracle seulement).

Créer des tables

Voici un exemple de création de table :

```
CREATE TABLE CLIENT(
        numcli int,
        nomcli varchar(32));
desc CLIENT;
```

Ajouter une ligne dans une table

Voici un exemple d'insertion de données dans une table :

```
INSERT INTO CLIENT (numcli, nomcli)
       VALUES (1, 'Marcel'), (2, 'Gégé');
SELECT * FROM CLIENT;
```

Attention, chaque commande SQL se termine par un point-virgule !

Suppression d'une table

Une table se supprime avec l'instruction **DROP TABLE**.

```
DROP TABLE CLIENT;
```

1.2 Contraintes déclaratives

1.2.1 Valeurs par défaut

```
create table client
        (
        numcli int,
        nom varchar(256) default 'Moi',
        prenom varchar(256)
        )
```

fait de 'Moi' le nom par défaut.

1.2.2 Champs non renseignés

```
create table client
        (
        numcli int,
        nom varchar(256) NOT NULL,
        prenom varchar(256) NOT NULL
        )
```

force la saisie des champs nom et prénom.

1.2.3 Clé primaire

Une clé primaire est :

- toujours renseignée
- unique

On peut préciser PRIMARY KEY dans la création de table

```
create table client
        (
        numcli int PRIMARY KEY,
        nom varchar(256),
        prenom varchar(256)
        )
```

La colonne numcli est clé primaire, toute insertion ne respectant pas la contrainte de clé primaire sera refusée par le SGBD.

1.2.4 Clé étrangère

Dans le cas où l'on souhaite garder en mémoire des factures émises par des clients, la façon de faire est de créer une deuxième table contenant la liste des factures :

```
create table facture
        (
        numfact int PRIMARY KEY,
        montantFacture int
        numcli int REFERENCES CLIENT(numCli)
        );
```

Le champ numCli dans cette table est clé étrangère, ce qui signifie qu'une ligne ne pourra être insérée dans la table facture que si le numcli de cette ligne existe dans la colonne numcli de la table client.

La syntaxe est

```
REFERENCES <nomtable> (<nomcolonne>)
```

1.2.5 Syntaxe alternative

Il est possible de définir les contraintes après la création d'une table.

```
ALTER TABLE nomtable
ADD [CONSTRAINT nomcontrainte] descriptioncontrainte;
```

descriptioncontrainte d'une clé primaire :

```
PRIMARY KEY(colonne1, ..., colonnen)
```

descriptioncontrainte d'une clé étrangère :

```
FOREIGN KEY(colonne1, ..., colonnen)
REFERENCES tablereferencee (colonne1, ..., colonnen)
```

Il est aussi possible de placer une descriptioncontrainte dans le `CREATE TABLE`. Par exemple,

```
create table facture
        (
        numfact int,
        montantFacture int
        numcli int,
        PRIMARY KEY (numfact),
        FOREIGN KEY nucli REFERENCES CLIENT(numcli)
        );
```

On remarque qu'il est possible de nommer une contrainte. C'est utile si on souhaite la supprimer :

```
ALTER TABLE nomtable DROP CONSTRAINT nomcontrainte;
```

Pour lister les contraintes sous Oracle, on utilise la commande :

```
SELECT * FROM USER_CONSTRAINTS;
```

Sous mySQL :

```
SHOW TABLE STATUS;
```

1.3 Introduction aux requêtes

1.3.1 Compléments sur SELECT

Il est possible d'utiliser SELECT pour n'afficher que certaines colonnes d'une table. Syntaxe :

```
SELECT <colonne_1>, <colonne_2>, ... , <colonne_n>
FROM <table>
```

Cette instruction s'appelle une requête, elle affichera pour chaque ligne de la table les valeurs des colonnes $colonne_1$ à $colonne_n$. Il est possible de supprimer les lignes en double à l'aide du mot-clé DISTINCT. Par exemple :

```
SELECT DISTINCT <colonne_1>, <colonne_2>, ... , <colonne_n>
FROM <table>
```

Pour trier les données, on utilise ORDER BY. Exemple :

```
SELECT <colonne_1>, <colonne_2>, ... , <colonne_n>
FROM <table>
ORDER BY <colonne_1bis>, <colonne_2bis>, ... , <colonne_nbis>
```

Cette instruction trie les données par $colonne_{1bis}$ croissants. En cas d'égalité, le tri est fait par $colonne_{2bis}$ croissants, etc. Pour trier par ordre décroissant, on ajoute DESC après le nom de la colonne choisie comme critère décroissant. Par exemple :

```
SELECT <colonne_1>, <colonne_2>, ... , <colonne_n>
FROM <table>
ORDER BY <colonne_1bis> DESC, <colonne_2bis>, ... , <colonne_nbis>
```

1.3.2 Instruction WHERE

Cette instruction permet de ne sélectionner que certaines lignes de la table. Par exemple la requête va afficher le nom du produit numéro 1 :

```
SELECT nomprod
FROM produit
WHERE numprod = 1
```

La syntaxe générale est

```
SELECT <colonne_1>, <colonne_2>, ... , <colonne_n>
FROM <table>
WHERE <condition>
```

condition sera évaluée pour chaque ligne de la table, et seules celles qui véfieront cette condition feront partie du résultat de la requête.

1.3.3 Conditions

Comparaison

Les conditions peuvent être des relations d'égalité (=), de différence (<>), d'inégalité (<, >, >= ou <=) sur des colonnes :

```
numero_client = 2
nom_client = 'Marcel'
prenom_client <> 'Ginette'
salary < 230
taxes >= 23000
```

Négation

La négation d'une condition s'obtient à l'aide de `NOT`. Par exemple, il est possible de ré-ecrire les conditions ci-avant :

```
NOT (numero_client <> 2)
NOT (nom_client <> 'Marcel')
NOT (prenom_client = 'Ginette')
NOT (salary >= 230)
NOT (taxes < 23000)
```

Connecteurs logiques

De même, vous avez à votre disposition tous les connecteurs logiques binaires : `AND`, `OR`. Ainsi, les deux conditions suivantes sont les mêmes :

```
NOT((nom = 'Raymond') AND (prenom <> 'Huguette'))
(nom <> 'Raymond') OR (prenom = 'Huguette')
```

`NULL`lité

Un champ non renseigné a la valeur `NULL`, dans une comparaison, `NULL` n'est jamais égal à quelque valeur qu'il soit ! La condition suivante est toujours fausse :

```
NULL = NULL;
```

La requête suivante ne renvoie aucune ligne :

```
INSERT INTO MP3 (numMp3) VALUES (3);
SELECT *
FROM MP3
WHERE nomMp3 = NULL;
```

Pour tester la nullité d'un champ, on utilise `IS NULL`, par exemple :

```
SELECT *
FROM MP3
WHERE nomMp3 IS NULL;
```

La non-nullité se teste de deux façons :

```
WHERE NOT (nomMp3 IS NULL);
```

ou encore

```
SELECT *
FROM MP3
WHERE nomMp3 IS NOT NULL;

numMp3  nomMp3
1       Get Lucky
2       Locked Down
```

Encadrement

Une valeur numérique peut être encadrée à l'aide de l'opérateur `BETWEEN`, par exemple les deux conditions suivantes sont équivalentes :

```
SALAIRE BETWEEN 1000 AND 5000
(SALAIRE >= 1000) AND (SALAIRE <= 5000)
```

Inclusion

L'opérateur `IN` permet de tester l'appartenance à une liste de valeurs. Les deux propositions suivantes sont équivalentes

```
NAME IN ('Gégé', 'Ginette', 'Marcel')
(NAME = 'Gégé') OR (NAME = 'Ginette') OR (NAME = 'Marcel')
```

LIKE

`LIKE` sert à comparer le contenu d'une variable à un littéral générique. Par exemple, la condition

```
NAME LIKE 'M%'
```

sera vérifiée si `NAME` commence par un `'M'`. Ca fonctionne aussi sur les valeurs de type numérique, la condition

```
SALARY LIKE '%000000000'
```

sera vérifiée si `SALARY` se termine par `000000000`. Le caractère `%` peut remplacer dans le littéral n'importe que suite, vide ou non, de caractères ; il a le même rôle que `*` en DOS et en SHELL. Le caractère `_` remplace un et un seul caractère dans le littéral. Par exemple, la condition

```
NAME LIKE 'K_r%'
```

ne sera vérifiée que si `NAME` commence par un `'K'` et contient un `'r'` en troisième position.

1.3.4 Suppression

L'expression

```
DELETE FROM <NOMTABLE>
WHERE <CONDITION>
```

efface de la table `NOMTABLE` toutes les lignes vérifiant `condition`. Attention ! La commande

```
DELETE FROM <NOMTABLE>
```

efface toutes les lignes de la table `NOMTABLE` !

1.3.5 Mise à jour

L'expression

```
UPDATE <NOMTABLE>
SET <colonne_1> = <valeur_1>,
<colonne_2> = <valeur_2>,
...,
<colonne_n> = <valeur_n>
WHERE <CONDITION>
```

modifie les lignes de la table `NOMTABLE` vérifiant `condition`. Elle affecte au champ $colonne_i$ la valeur $valeur_i$. Par exemple,

```
UPDATE CLIENT
SET prenomcli = 'Dark'
WHERE nomcli = 'Vador'
```

affecte la valeur `'Dark'` aux champs `prenomcli` de toutes les lignes dont la valeur `nomcli` est égale à `'Vador'`. Il est possible, dans une modification, d'utiliser les valeurs des autres champs de la ligne, voire même l'ancienne valeur de ce champ. Par exemple,

```
UPDATE OPERATION
SET montantoper = montantoper + 5000
```

augmente les montants de toutes les opérations bancaires de 5000 (choisissez l'unité !).

1.4 Jointures

1.4.1 Principe

Nous utiliserons pour ce cours les données de .3.

Si on souhaite connaître les numéros des modules pré-requis pour s'inscrire dans le module 'PL/SQL Oracle', il nous faut tout d'abord le numéro de ce module :

```
SELECT numMod
FROM MODULE
WHERE nomMod = 'PL/SQL Oracle'

numMod
6
```

Ensuite, cherchons les numéros des modules pré-requis pour s'inscrire dans le module numéro 6,

```
SELECT numModPrereq
FROM PREREQUIS
WHERE numMod = 6;

numModPrereq
1
5
```

Et pour finir, allons récupérer les noms de ces modules,

```
SELECT nomMod
FROM MODULE
WHERE numMod IN (1, 5);

nomMod
Oracle
Merise
```

Vous êtes probablement tous en train de vous demander s'il n'existe pas une méthode plus simple et plus rapide, et surtout une façon d'automatiser ce que nous venons de faire. Il existe un moyen de sélectionner des données dans plusieurs tables simultanément. Pour traiter la question ci-dessus il suffisait de saisir :

```
SELECT m2.nomMod
FROM MODULE m1, MODULE m2, PREREQUIS p
WHERE m1.numMod = p.numMod
AND m2.numMod = p.numModprereq
AND m1.nomMod = 'PL/SQL Oracle';

nomMod
Oracle
Merise
```

Le but de ce chapitre est d'expliciter ce type de commande.

1.4.2 Produit cartésien

L'instruction `SELECT ... FROM ...` peut s'étendre de la façon suivante :

```
SELECT <listecolonnes>
FROM <listetables>
```

L'exemple ci-dessous vous montre le résultat d'une telle commande.

```
SELECT *
FROM PROPOSER, PRODUIT;

numfou   numprod  prix    numprod  nomprod
1        1        2       1        Bocal de cornichons
2        1        3       1        Bocal de cornichons
2        2        2       1        Bocal de cornichons
2        3        1       1        Bocal de cornichons
1        1        2       2        Tube de dentifrice
2        1        3       2        Tube de dentifrice
2        2        2       2        Tube de dentifrice
2        3        1       2        Tube de dentifrice
1        1        2       3        Flacon de lotion anti-escarres
2        1        3       3        Flacon de lotion anti-escarres
2        2        2       3        Flacon de lotion anti-escarres
2        3        1       3        Flacon de lotion anti-escarres
1        1        2       4        Déodorant fraîcheur 96 heures
2        1        3       4        Déodorant fraîcheur 96 heures
2        2        2       4        Déodorant fraîcheur 96 heures
2        3        1       4        Déodorant fraîcheur 96 heures
```

Placer une liste de tables dans le `FROM` revient à former toutes les combinaisons de lignes possibles. Cependant, cela a relativement peu de sens.

1.4.3 Jointure

Il serait plus intéressant, dans le cas présent, de ne voir s'afficher que des lignes dont les numéros de produits concordent. Pour ce faire, il suffit d'utiliser `WHERE`. Par exemple,

```
SELECT *
FROM PROPOSER, PRODUIT
WHERE PROPOSER.numprod = PRODUIT.numprod;

numfou   numprod  prix    numprod  nomprod
1        1        2       1        Bocal de cornichons
2        1        3       1        Bocal de cornichons
2        2        2       2        Tube de dentifrice
2        3        1       3        Flacon de lotion anti-escarres
```

Nous avons mis en correspondance des lignes de la table `proposer` avec des lignes de la table `produit` en utilisant le fait que `numprod` est une clé étrangère dans `proposer`. Comme la colonne `numprod` apparaît deux fois dans la requête, il est nécessaire de la préfixer par le nom de la table de sorte que chaque colonne puisse être désignée de façon non ambiguë. Si on veut mettre face à face les noms des produits et les noms des fournisseurs, il suffit de saisir la requête

```
SELECT nomfou, nomprod
FROM PRODUIT, FOURNISSEUR, PROPOSER
WHERE PRODUIT.numProd = PROPOSER.numProd
AND FOURNISSEUR.numFou = PROPOSER.numFou;

nomfou   nomprod
Bocaux Gérard   Bocal de cornichons
Paramédical Gisèle      Bocal de cornichons
Paramédical Gisèle      Tube de dentifrice
Paramédical Gisèle      Flacon de lotion anti-escarres
```

1.4.4 Jointures réflexives

En utilisant la syntaxe suivante, il est possible de renommer les tables,

```
FROM <table_1> <table_1_renommee>, ..., <table_n> <table_n_renommee>
```

Reformulons la requête ci-dessus,

```
SELECT nomfou, nomprod
FROM PRODUIT p, FOURNISSEUR f, PROPOSER pr
WHERE p.numProd = pr.numProd
AND f.numFou = pr.numFou;

nomfou  nomprod
Bocaux Gérard   Bocal de cornichons
Paramédical Gisèle      Bocal de cornichons
Paramédical Gisèle      Tube de dentifrice
Paramédical Gisèle      Flacon de lotion anti-escarres
```

Le renommage permet entre autres de faire des jointures réflexives, c'est à dire entre une table et elle même. Par exemple, en reprenant la table intervalle,

```
SELECT * FROM INTERVALLE;

borneInf        borneSup
0       30
2       3
2       56
5       10
7       32
8       27
12      3
12      30
21      8
34      26
```

La commande ci-dessous affiche tous les couples d'intervalles ayant une borne en commun,

```
SELECT * FROM INTERVALLE i, INTERVALLE j
WHERE (i.borneInf = j.borneInf AND i.borneSup < j.borneSup)
OR (i.borneInf < j.borneInf AND i.borneSup = j.borneSup);

borneInf        borneSup        borneInf        borneSup
0       30      12      30
2       3       2       56
2       3       12      3
12      3       12      30
```

1.5 Agrégation de données

1.5.1 Fonctions d'agrégation

Exemple introductif

Nous voulons connaître le nombre de lignes de table *produit*. Deux façons de procéder :

1. **Solution moche**

```
SELECT * FROM PRODUIT;

numprod nomprod
1       Bocal de cornichons
2       Tube de dentifrice
3       Flacon de lotion anti-escarres
4       Déodorant fraîcheur 96 heures
```

   On a la réponse avec le nombre de lignes sélectionnées.

2. **Solution belle**

```
SELECT count(*) FROM PRODUIT;

count(*)
4
```

   La réponse est le résultat de la requête.

Définition

Une fonction d'agrégation retourne une valeur calculée sur toutes les lignes de la requête (nombre, moyenne...). Nous allons utiliser les suivantes :

- `COUNT(col)` : retourne le nombre de lignes dont le champ `col` est non `NULL`.
- `AVG(col)` : retourne la moyenne des valeurs `col` sur toutes les lignes dont le champ `col` est non `NULL`.
- `MAX(col)` : retourne la plus grande des valeurs `col` sur toutes les lignes dont le champ `col` est non `NULL`.
- `MIN(col)` : retourne la plus petite des valeurs `col` sur toutes les lignes dont le champ `col` est non `NULL`.
- `SUM(col)` : retourne la somme des valeurs `col` sur toutes les lignes dont le champ `col` est non `NULL`.

Exemples d'utilisation

L'exemple suivant retourne le prix du produit proposé au prix maximal.

```
SELECT MAX(prix)
FROM PROPOSER;

MAX(prix)
3
```

Il est possible de renommer la colonne `MAX(prix)`, en utilisant le mot clé `AS` :

```
SELECT MAX(prix) AS PRIX_MAXIMAL
FROM PROPOSER;

PRIX_MAXIMAL
3
```

Les requêtes suivantes récupèrent le nom du fournisseur proposant l'article 'Bocal de cornichons' au prix le moins élevé :

```sql
SELECT MIN(prix) AS PRIX_MINIMUM
FROM PROPOSER PR, PRODUIT P
WHERE PR.numprod = P.numprod
AND nomprod = 'Bocal de cornichons';

PRIX_MINIMUM
2
```

```sql
SELECT nomfou
FROM FOURNISSEUR F, PROPOSER PR, PRODUIT P
WHERE F.numfou = PR.numfou
AND PR.numprod = P.numprod
AND nomprod = 'Bocal de cornichons'
AND prix = 1;
```

Il est possible de faire cela avec une seule requête en récupérant le prix minimum dans une requête imbriquée. Mais cela sera pour un cours ultérieur.

Compléments sur COUNT

On récupère le nombre de ligne retournées par une requête en utilisant COUNT(*). Par exemple, si on souhaite connaître le nombre de produits proposés par le fournisseur 'Bocaux Gérard' :

```sql
SELECT COUNT(*) AS NB_PROD
FROM FOURNISSEUR F, PROPOSER P
WHERE F.numfou = P.numfou
AND nomfou = 'Bocaux Gérard';

NB_PROD
1
```

On aurait aussi pu saisir :

```sql
SELECT COUNT(numprod) AS NB_PROD
FROM FOURNISSEUR F, PROPOSER P
WHERE F.numfou = P.numfou
AND nomfou = 'Bocaux Gérard';

NB_PROD
1
```

Pour connaître le nombre de produits proposés, c'est à dire dont le numprod a une occurence dans la table PROPOSER, on procède de la façon suivante :

```sql
SELECT COUNT(DISTINCT numprod) AS NB_PRODUITS_PROPOSES
FROM PROPOSER;

NB_PRODUITS_PROPOSES
3
```

Le DISTINCT nous sert à éviter qu'un même produit proposé par des fournisseurs différents soit comptabilisé plusieurs fois.

1.5.2 Groupage

L'instruction GROUP BY

Les opérations d'agrégation considérées jusqu'à maintenant portent sur la totalité des lignes retournées par les requêtes, l'instruction GROUP BY permet de former des paquets à l'intérieur desquels les données seront agrégées. Cette instruction s'utilise de la manière suivante

```sql
SELECT ...
FROM ...
WHERE...
GROUP BY <liste_colonnes>
ORDER BY ...
```

La liste des colonnes sert de critère pour répartir les lignes dans des paquets de lignes. Si par exemple nous souhaitons afficher la liste des nombres de produits proposés par chaque fournisseur :

```sql
SELECT nomfou, COUNT(*) AS NB_PRODUITS_PROPOSES
FROM FOURNISSEUR F, PROPOSER P
WHERE F.numfou = P.numfou
GROUP BY nomfou;

nomfou  NB_PRODUITS_PROPOSES
Bocaux Gérard   1
Paramédical Gisèle      3
```

L'instruction `HAVING`

Supposons que de la requête précédente, nous ne souhaitions garder que les lignes pour lesquelles la valeur `NB_PRODUITS_PROPOSES` est égale à 1. Ajouter une condition dans `WHERE` serait inutile, le filtrage occasionné par `WHERE` est effectué avant l'agrégation. Il nous faudrait une instruction pour n'inclure que des groupes de données répondant certains critères. L'instruction utilisée pour ce faire est `HAVING`. Son utilisation est la suivante :

```sql
SELECT ...
FROM ...
WHERE ...
GROUP BY...
HAVING <condition>
ORDER BY ...
```

Par exemple,

```sql
SELECT nomfou, COUNT(numprod) AS NB_PRODUITS_PROPOSES
FROM FOURNISSEUR F, PROPOSER P
WHERE F.numfou = P.numfou
GROUP BY nomfou
HAVING COUNT(numprod) = 1
ORDER BY nomfou DESC;

nomfou  NB_PRODUITS_PROPOSES
Bocaux Gérard   1
```

Affichons les noms des fournisseurs qui ont livré strictement plus d'un produit différent (toutes livraisons confondues),

```sql
SELECT nomfou
FROM FOURNISSEUR F, DETAILLIVRAISON D
WHERE F.numfou = D.numfou
GROUP BY F.numfou, nomfou
HAVING count(DISTINCT numprod) > 1;

nomfou
Paramédical Gisèle
```

1.6 Vues

1.6.1 Définition

Une vue est une table contenant des données calculées sur celle d'une autre table. Les données d'une vue sont tout le temps à jour. Si vous modifiez les données d'une des tables sur lesquelles est calculée la vue, alors les modifications sont automatiquement répercutées sur la vue.

1.6.2 Syntaxe

Appréciez la simplicité de la syntaxe :

```
CREATE VIEW <nom_vue> AS <requete>
```

1.6.3 Application

Par exemple, la requête suivante met en correpondance les noms des produits avec le nombre de fournisseurs qui le proposent :

```
SELECT nomprod, COUNT(numfou) AS NB_FOURNISSEURS
FROM PRODUIT P
        LEFT OUTER JOIN PROPOSER PR
        ON P.numprod = PR.numprod
GROUP BY nomprod
ORDER BY COUNT(numfou);

nomprod NB_FOURNISSEURS
Déodorant fraîcheur 96 heures   0
Tube de dentifrice       1
Flacon de lotion anti-escarres  1
Bocal de cornichons      2
```

Ce type de requête sera explité dans un cours ultérieur. Pour le moment, notez juste que les outils dont vous disposez pour le moment ne vous permettront pas de formuler une requête affichant les noms des produits n'ayant aucun fournisseur. Créons une vue pour ne pas avoir à se farcir la requête chaque fois que nous aurons besoin de ces informations :

```
CREATE VIEW NB_FOURNISSEURS_PAR_PRODUIT AS
SELECT nomprod, COUNT(numfou) AS NB_FOURNISSEURS
FROM PRODUIT P
        LEFT OUTER JOIN PROPOSER PR
        ON P.numprod = PR.numprod
GROUP BY nomprod
ORDER BY COUNT(numfou);
```

Une fois créée, on peut interroger une vue de la même façon qu'on interroge une table :

```
SELECT *
FROM NB_FOURNISSEURS_PAR_PRODUIT;

nomprod NB_FOURNISSEURS
Déodorant fraîcheur 96 heures   0
Tube de dentifrice       1
Flacon de lotion anti-escarres  1
Bocal de cornichons      2
```

Notez que toute modification dans la table `PROPOSER` ou `PRODUIT` sera immédiatement répercutée sur la vue.

```
INSERT INTO PROPOSER VALUES (3, 4, 9);
SELECT *
```

```
FROM NB_FOURNISSEURS_PAR_PRODUIT;

nomprod NB_FOURNISSEURS
Tube de dentifrice      1
Flacon de lotion anti-escarres  1
Déodorant fraîcheur 96 heures   1
Bocal de cornichons     2
```

```
INSERT INTO PROPOSER VALUES (3, 4, 9);
SELECT *
FROM NB_FOURNISSEURS_PAR_PRODUIT;
```

Maintenant, nous souhaitons voir s'afficher, pour tout i, le nombre de produits proposés par exactement i fournisseurs.

```
SELECT CONCAT ('Il y a ', COUNT(*), ' produit(s) qui est/sont proposé(s) par ',
        NB_FOURNISSEURS, ' fournisseur(s).')
        AS NOMBRE_DE_FOURNISSEURS
FROM NB_FOURNISSEURS_PAR_PRODUIT
GROUP BY NB_FOURNISSEURS
ORDER BY NB_FOURNISSEURS;

NOMBRE_DE_FOURNISSEURS
Il y a 3 produit(s) qui est/sont proposé(s) par 1 fournisseur(s).
Il y a 1 produit(s) qui est/sont proposé(s) par 2 fournisseur(s).
```

1.6.4 Suppression

On supprime une vue avec l'instruction suivante :

```
DROP VIEW <nom_vue>;
```

1.7 Requêtes imbriquées

Oracle permet d'imbriquer les requêtes, c'est-à-dire de placer des requêtes dans les requêtes. Une requête imbriquée peut renvoyer trois types de résultats :

- une valeur scalaire
- une colonne
- une table

1.7.1 Sous requêtes renvoyant une valeur scalaire

Le résultat d'une requête est dit scalaire s'il comporte une seule ligne et une seule colonne. Par exemple :

```
SELECT COUNT(*) FROM PRODUIT

COUNT(*)
4
```

On peut placer dans une requête une sous-requête calculant un résultat scalaire. Un tel type de sous-requête se place soit comme une colonne supplémentaire, soit comme une valeur servant à évaluer des conditions (`WHERE` ou `HAVING`).

Colonne fictive

On peut ajouter une colonne dans une requête, et choisir comme valeurs pour cette colonne le résultat d'une requête. Ce type de requête est souvent une alternative à `GROUP BY`. Par exemple, la requête suivante nous renvoie, pour tout produit, le nombre de fournisseurs proposant ce produit :

```
SELECT nomprod, (SELECT COUNT(*)
        FROM PROPOSER PR
        WHERE PR.numprod = P.numprod) AS NB_FOURNISSEURS
FROM PRODUIT P

nomprod NB_FOURNISSEURS
Bocal de cornichons      2
Tube de dentifrice       1
Flacon de lotion anti-escarres   1
Déodorant fraîcheur 96 heures    0
```

Conditions complexes

On peut construire une condition en utilisant le résultat d'une requête. Pour notre exemple, déclarons d'abord une vue contenant le nombre d'articles proposés par chaque fournisseur,

```
CREATE VIEW NB_PROD_PAR_FOU AS
        SELECT numfou, (SELECT COUNT(*)
                FROM PROPOSER P
                WHERE P.numfou = F.numfou) AS NB_PROD
        FROM FOURNISSEUR F
```

Ensuite, recherchons les noms des fournisseurs proposant le plus de produits :

```
SELECT nomfou
FROM FOURNISSEUR F, NB_PROD_PAR_FOU N
WHERE F.numfou = N.numfou
AND NB_PROD = (SELECT MAX(NB_PROD)
        FROM NB_PROD_PAR_FOU)

nomfou
Paramédical Gisèle
```

La requête `SELECT MAX(NB_PROD) FROM NB_PROD_PAR_FOU` est évaluée avant, et son résultat lui est substitué dans l'expression de la requête. Comme on a

```
SELECT MAX(NB_PROD) FROM NB_PROD_PAR_FOU;

MAX(NB_PROD)
3
```

Alors la requête précédente, dans ce contexte, est équivalente à

```
SELECT nomfou
FROM FOURNISSEUR F, NB_PROD_PAR_FOU N
WHERE F.numfou = N.numfou
AND NB_PROD = 2
```

`INSERT` et `UPDATE`

On peut placer dans des instructions de mises à jour ou d'insertions des requêtes imbriquées. Par exemple,

```
INSERT INTO PERSONNE (numpers, nom, prenom)
VALUES ((SELECT MAX(numpers) + 1 FROM PERSONNE),
'Darth', 'Vador');
```

1.7.2 Sous requêtes renvoyant une colonne

On considère une colonne comme une liste de valeurs, on peut tester l'appartance d'un élément à cette liste à l'aide de l'opérateur `IN`. On peut s'en servir comme une alternative aux jointures, par exemple, réécrivons la requête de la section précédente. La requête suivante nous renvoie le nombre de produits proposés par les fournisseurs proposant le plus de produits :

```
SELECT MAX(NB_PROD) FROM NB_PROD_PAR_FOU

MAX(NB_PROD)
3
```

Maintenant, recherchons les numéros des fournisseurs proposant un tel nombre de produits :

```
SELECT N.numfou
FROM NB_PROD_PAR_FOU N
WHERE NB_PROD = (SELECT MAX(NB_PROD)
        FROM NB_PROD_PAR_FOU)

numfou
2
```

Notons que s'il existe plusieurs fournisseurs proposant 2 produits, cette requête renverra plusieurs lignes. C'est donc par hasard qu'elle ne retourne qu'une ligne. Le numéro du fournisseur proposant le plus de produits est donc le 1. Cherchons ce fournisseur :

```
SELECT nomfou
FROM FOURNISSEUR F
WHERE F.numfou IN (1)

nomfou
Bocaux Gérard
```

Il suffit donc dans la requête ci-dessous de remplacer le 1 par la requête qui a retourné 1. On a finalement :

```
SELECT nomfou
FROM FOURNISSEUR F
WHERE F.numfou IN (SELECT N.numfou
        FROM NB_PROD_PAR_FOU N
        WHERE NB_PROD = (SELECT MAX(NB_PROD)
                FROM NB_PROD_PAR_FOU))

nomfou
Paramédical Gisèle
```

1.7.3 Sous requêtes non correlées renvoyant une table

On peut remplacer le nom d'une table dans la clause `FROM` par une sous-requête. Par exemple, la requête suivante renvoie une table.

```
SQL> SELECT
  2     (SELECT COUNT(*)
  3         FROM PROPOSER PR
  4        WHERE PR.numfou = F.numfou
  5     ) AS NB_PROD
  6  FROM FOURNISSEUR F;

   NB_PROD
----------
         2
         1
         1
         0
```

Cette table contient, pour chaque fournisseur, le nombre de produits proposés. Si l'on souhaite connaître le plus grand nombre de produits proposés, on se sert du résultat de la requête ci-dessus comme d'une table :

```
SQL>  SELECT MAX(NB_PROD) AS MAX_NB_PROD
  2   FROM
  3     (SELECT
  4          (SELECT COUNT(*)
  5           FROM PROPOSER PR
  6           WHERE PR.numfou = F.numfou
  7          ) AS NB_PROD
  8      FROM FOURNISSEUR F
  9        );

MAX_NB_PROD
-----------
          2
```

Ce type de requête est une alternative aux vues. Récupérons maintenant les noms des fournisseurs proposant le plus de produits (sans jointure et sans vue !) :

```
SQL> SELECT nomfou
  2  FROM FOURNISSEUR
  3  WHERE numfou IN
  4     (SELECT numfou
  5      FROM
  6       (SELECT numfou,
  7            (SELECT COUNT(*)
  8             FROM PROPOSER PR
  9             WHERE PR.numfou = F.numfou
 10            ) AS NB_PROD
```

```
11         FROM FOURNISSEUR F
12         ) N
13       WHERE NB_PROD =
14         (SELECT MAX(NB_PROD)
15          FROM
16             (SELECT numfou,
17                   (SELECT COUNT(*)
18                    FROM PROPOSER PR
19                         WHERE PR.numfou = F.numfou
20                                 ) AS NB_PROD
21                               FROM FOURNISSEUR F
22             ) N
23          )
24       );

NOMFOU
------------------------------
f1
```

Vous constatez que la solution utilisant les vues est nettement plus simple.

1.7.4 Sous requêtes correlées

Une sous-requête peut être de deux types :
- **simple** : Elle évaluée avant la requête principale
- **correlée** : Elle est évaluée pour chaque ligne de la requête principale

Par exemple, la requête suivante renvoie le nombre de produits livrés pour chaque fournisseur. Elle contient une sous-requête correlée.

```
SQL> SELECT numfou,
  2      (SELECT SUM(qte)
  3       FROM DETAILLIVRAISON D
  4       WHERE D.numfou = F.numfou
  5         ) NB_PROD_L
  6  FROM FOURNISSEUR F;

    NUMFOU  NB_PROD_L
---------- ----------
         1         45
         2
         3         10
         4
```

Cette même requête, une fois évaluée, peut server de requête non correlée si on souhaite connaître les noms de ces fournisseurs :

```
SQL> SELECT nomfou, NB_PROD_L
  2  FROM FOURNISSEUR F,
  3       (SELECT numfou,
  4          (SELECT SUM(qte)
  5           FROM DETAILLIVRAISON D
  6           WHERE D.numfou = F.numfou
  7          ) NB_PROD_L
  8        FROM FOURNISSEUR F
  9        ) L
 10  WHERE F.numfou = L.numfou;

NOMFOU                          NB_PROD_L
------------------------------ ----------
f1                                     45
```

```
f2
f3                                       10
f4
```

Amusons-nous : quel sont, pour chaque fournisseur, les produits qui ont été les plus livrés ?

```
SQL> SELECT nomfou, nomprod
  2  FROM FOURNISSEUR F, PRODUIT P,
  3     (SELECT FF.numfou, PP.numprod
  4     FROM FOURNISSEUR FF, PRODUIT PP
  5        WHERE
  6          (SELECT SUM(qte)
  7          FROM DETAILLIVRAISON L
  8          WHERE L.numfou = FF.numfou
  9          AND L.numprod = PP.numprod
 10          )
 11          =
 12          (SELECT MAX(NB_PROD_L)
 13          FROM
 14            (SELECT numfou, SUM(qte) AS NB_PROD_L
 15            FROM DETAILLIVRAISON L
 16            GROUP BY numprod, numfou
 17            ) Q
 18          WHERE Q.numfou = FF.numfou
 19          )
 20     GROUP BY numfou, numprod
 21     ) M
 22  WHERE M.numprod = P.numprod
 23  AND M.numfou = F.numfou;

NOMFOU                          NOMPROD
------------------------------  ------------------------------
f1                              Roue de secours
f3                              Cotons tiges
```

Dans la requête précédente, quelles sous-requêtes sont correlées et lesquelles ne le sont pas ?

1.8 Procédures stockées

1.8.1 Exemple

Étant données la base de données de [.11]. Nous comptons implémenter les contraintes suivantes.

— Un exemplaire non empruntable ne peut pas être emprunté.
— Un exemplaire ne peut pas être en possession de deux adhérents à la fois.
— Un adhérent ne peut pas être en possession de deux exemplaires différents d'un même ouvrage.
— Un adhérent ne peut pas emprunter si son abonnement n'est pas à jour.
— Un adhérent ne peut pas être en possession de plus de cinq livres.
— Une même personne ne peut pas être à la fois cataloguée comme auteur et adhérent.

Toutes ces contraintes ne sont pas *déclaratives*, ce qui signifie qu'il est impossible dans le **create table** de les prendre en compte.

Comment faire ?

1.8.2 SQL Procédural

Le SQL procédural est une extension impérative de SQL. Elle permet d'exécuter des instructions à l'intérieur du serveur de base de données. Par exemple,

```
delimiter $$

drop procedure compteARebours;

create  procedure compteARebours(i integer)
begin
        declare j integer;
        if i >= 0 then
           set j = i;
           while j >= 0 do
                 select j;
                 set j = j - 1;
           end while;
        end if;
end;

call compteARebours(3);
$$

delimiter ;
```

1.8.3 Procédures

Il est possible de stocker des procédures de la même façon que dans les langages impératifs.

```
delimiter $$

drop procedure insertAdherent;
create procedure insertAdherent (nom varchar(64), prenom varchar(64), mail varchar(64))
begin
        insert into personne (nompers, prenompers) values (nom, prenom);
        insert into adherent (numpers, mailadherent) values (last_insert_id(), mail);
end
$$

call insertAdherent('Morflegroin', 'Marcel', 'marcel@morflegroin.com');
call insertAdherent('Le Ballon', 'Gégé', 'gege.m@grosbuveur.com');
call insertAdherent('Couledru', 'Gertrude', 'g.proflechettes@ligue-flechettes.fr');
```

```
$$

delimiter ;

insert into personne (nompers, prenompers) values ('Rowlings', 'J. K.');

delimiter $$

drop procedure insertOuvrage;
create procedure insertOuvrage (titre varchar(64), numAuteur integer, nombreExemplaires
    integer)
begin
        declare ouvrage_inserted_id integer;
        declare i integer;
        insert into ouvrage(numauteur, titreouvrage) values (numAuteur, titre);
        set ouvrage_inserted_id = last_insert_id();
        set i = 1;
        while i <= nombreExemplaires do
              insert into exemplaire (numOuvrage, numExemplaire) values (ouvrage_inserted_id
                  , i);
              set i = i + 1;
        end while;
end
$$

delimiter ;

call insertOuvrage('Harry Potter and the Deathly Hallows', 4, 10);
```

1.8.4 Curseurs

Un curseur permet de parcourir une à une les lignes résultant d'un SELECT.

```
delimiter $$

DROP PROCEDURE IF EXISTS AfficheUtilisateurs;
CREATE PROCEDURE AfficheUtilisateurs()
BEGIN
        DECLARE num_pers integer;
        DECLARE nom_pers varchar(64);
        DECLARE prenom_pers varchar(64);
        DECLARE nb_a integer;
        DECLARE finished boolean DEFAULT FALSE;
        DECLARE personnes CURSOR FOR SELECT numpers, nompers, prenompers FROM personne;
        DECLARE CONTINUE HANDLER FOR NOT FOUND SET finished = TRUE;

        OPEN personnes;
        personnesloop: LOOP
               FETCH personnes INTO num_pers, nom_pers, prenom_pers;
               IF finished THEN
                  LEAVE personnesloop;
               END IF;
               SELECT COUNT(*) INTO nb_a FROM adherent WHERE numpers = num_pers;
               IF nb_a > 0 THEN
                  select concat(prenom_pers, ' ', nom_pers, ' est un adherent');
               ELSE
                  select concat(prenom_pers, ' ', nom_pers, ' est un auteur');
               END IF;
        END LOOP;
```

```
        CLOSE personnes;
END;
$$

CALL AfficheUtilisateurs();
$$

delimiter ;
```

1.8.5 Triggers

Un trigger est une procédure stockée de déclenchant automatiquement à la suite d'un événement.

```
delimiter $$

drop trigger if exists adherentBeforeInsert;
create trigger adherentBeforeInsert before insert on adherent for each row
begin
        declare nb_aut integer;
        declare error_msg varchar(128);
        /* Met la date système par défaut  */
        if new.daterenouvellement is null then
           set new.daterenouvellement = now();
        end if;
        /* Vérifie que l'adhérent n'est pas déjà un auteur */
        select count(*) into nb_aut
               from ouvrage
               where new.numpers = numauteur;
        if nb_aut > 0 then
           set error_msg = concat('L\'adhérent ', new.numpers, 'est déjà un auteur.');
           signal sqlstate '45000' set message_text = error_msg;
        end if;
end;
$$

delimiter ;
```

.1 Scripts de création de tables

.2 Livraisons Sans contraintes

Attention : Le numéro de livraison est une clé secondaire, c'est-à-dire un numéro unique étant donné un fournisseur.

```
drop table PRODUIT;
drop table FOURNISSEUR;
drop table PROPOSER;
drop table LIVRAISON;
drop table DETAILLIVRAISON;

CREATE TABLE PRODUIT
(numprod integer,
nomprod varchar(30));

CREATE TABLE FOURNISSEUR
(numfou integer,
nomfou varchar(30));

CREATE TABLE PROPOSER
(numfou integer,
numprod integer,
prix real);

CREATE TABLE LIVRAISON
(numfou integer,
numli integer,
dateli date
);

CREATE TABLE DETAILLIVRAISON
(numfou integer,
numli integer,
numprod integer,
qte integer);
```

.3 Modules et prerequis

les modules sont répertoriés dans une table, et les modules pré-requis pour s'y inscrire (avec la note minimale) se trouvent dans la table prerequis. Une ligne de la table `PREREQUIS` nous indique que pour s'inscrire dans le module numéro `numMod`, il faut avoir eu au moins `noteMin` au module `numModPrereq`.

```
DROP TABLE IF EXISTS RESULTAT;
DROP TABLE IF EXISTS EXAMEN;
DROP TABLE IF EXISTS PREREQUIS;
DROP TABLE IF EXISTS INSCRIPTION;
DROP TABLE IF EXISTS MODULE;
DROP TABLE IF EXISTS ETUDIANT;

CREATE TABLE ETUDIANT
        (numEtud int PRIMARY KEY,
        nom varchar(40),
        prenom varchar(40),
        datenaiss date,
        civilite varchar(4),
        patronyme varchar(40),
        numsecu varchar(15) NOT NULL
```

```sql
        );

CREATE TABLE MODULE
        (numMod int PRIMARY KEY,
        nomMod varchar(15),
        effecMax int DEFAULT 30
        );

CREATE TABLE EXAMEN
        (numMod int REFERENCES MODULE(numMod),
        numExam int,
        dateExam date,
        PRIMARY KEY(numMod, numExam)
        );

CREATE TABLE INSCRIPTION
        (numEtud int REFERENCES ETUDIANT(numEtud),
        numMod int REFERENCES MODULE(numMod),
        dateInsc date,
        PRIMARY KEY(numEtud, numMod)
        );

CREATE TABLE PREREQUIS
        (numMod int REFERENCES MODULE(numMod),
        numModPrereq int REFERENCES MODULE(numMod),
        noteMin int NOT NULL DEFAULT 10,
        PRIMARY KEY(numMod, numModPrereq)
        );

CREATE TABLE RESULTAT
        (numMod int,
        numExam int,
        numEtud int,
        note int,
        PRIMARY KEY(numMod, numExam, numEtud),
        FOREIGN KEY (numMod, numExam) REFERENCES EXAMEN(numMod, numExam),
        FOREIGN KEY (numEtud, numMod) REFERENCES INSCRIPTION(numEtud, numMod)
        );

INSERT INTO MODULE (numMod, nomMod) VALUES
(1, 'Oracle'),
(2, 'C++'),
(3, 'C'),
(4, 'Algo'),
(5, 'Merise'),
(6, 'PL/SQL Oracle'),
(7, 'mySQL'),
(8, 'Algo avancee');

INSERT INTO PREREQUIS (numMod, numModPrereq) VALUES
(1, 5),
(2, 3),
(6, 5),
(8, 5),
(7, 5);
INSERT INTO PREREQUIS VALUES (6, 1, 12);
```

.4 Géométrie

La table `INTERVALLE` contient des intervalles spécifiés par leurs bornes inférieure et supérieure. Supprimer de la table intervalle tous les intervalles qui n'en sont pas avec une seule instruction.

```
drop table if exists RECTANGLE ;
drop table if exists INTERVALLE;

CREATE TABLE INTERVALLE
(borneInf int,
 borneSup int,
 PRIMARY KEY (borneInf, borneSup));

CREATE TABLE RECTANGLE
(xHautGauche int,
 yHautGauche int,
 xBasDroit int,
 yBasDroit int,
 PRIMARY KEY (xHautGauche, yHautGauche, xBasDroit, yBasDroit));

INSERT INTO INTERVALLE VALUES (2, 56);
INSERT INTO INTERVALLE VALUES (12, 30);
INSERT INTO INTERVALLE VALUES (2, 3);
INSERT INTO INTERVALLE VALUES (12, 3);
INSERT INTO INTERVALLE VALUES (8, 27);
INSERT INTO INTERVALLE VALUES (34, 26);
INSERT INTO INTERVALLE VALUES (5, 10);
INSERT INTO INTERVALLE VALUES (7, 32);
INSERT INTO INTERVALLE VALUES (0, 30);
INSERT INTO INTERVALLE VALUES (21, 8);

INSERT INTO RECTANGLE VALUES (2, 12, 5, 7);
INSERT INTO RECTANGLE VALUES (2, 12, 1, 13);
INSERT INTO RECTANGLE VALUES (10, 13, 1, 11);
INSERT INTO RECTANGLE VALUES (10, 13, 10, 11);
INSERT INTO RECTANGLE VALUES (2, 7, 5, 13);
INSERT INTO RECTANGLE VALUES (21, 73, 15, 22);
INSERT INTO RECTANGLE VALUES (1, 2, 3, 4);
INSERT INTO RECTANGLE VALUES (1, 5, 3, 2);
INSERT INTO RECTANGLE VALUES (1, 6, 3, 6);
INSERT INTO RECTANGLE VALUES (4, 2, 1, 4);
INSERT INTO RECTANGLE VALUES (2, 3, 4, 0);
INSERT INTO RECTANGLE VALUES (5, 4, 2, 1);
```

.5 Livraisons

```
drop table if exists DETAILLIVRAISON;
drop table if exists LIVRAISON;
drop table if exists PROPOSER;
drop table if exists FOURNISSEUR;
drop table if exists PRODUIT;

CREATE TABLE PRODUIT
(numprod int,
nomprod varchar(30));

CREATE TABLE FOURNISSEUR
(numfou int,
```

```sql
nomfou varchar(30));

CREATE TABLE PROPOSER
(numfou int,
numprod int,
prix int NOT NULL);

CREATE TABLE LIVRAISON
(numfou int,
numli int,
dateli date
);

CREATE TABLE DETAILLIVRAISON
(numfou int,
numli int,
numprod int,
qte int NOT NULL);

 alter table PRODUIT add constraint pk_produit
PRIMARY KEY (numprod);
 alter table FOURNISSEUR add constraint pk_fournisseur
PRIMARY KEY (numfou);
 alter table PROPOSER add constraint pk_proposer
PRIMARY KEY (numfou, numprod);
 alter table LIVRAISON add constraint pk_livraison
PRIMARY KEY (numfou, numli);
 alter table DETAILLIVRAISON add constraint pk_detail_livraison
PRIMARY KEY (numfou, numli, numprod);
 alter table PROPOSER add constraint fk_proposer_fournisseur
FOREIGN KEY (numfou) REFERENCES FOURNISSEUR (numfou);
 alter table PROPOSER add constraint fk_proposer_produit
FOREIGN KEY (numprod) REFERENCES PRODUIT (numprod);
 alter table LIVRAISON add constraint fk_livraison
FOREIGN KEY (numfou) REFERENCES FOURNISSEUR (numfou);
 alter table DETAILLIVRAISON add constraint fk_detail_livraison
FOREIGN KEY (numfou, numli) REFERENCES LIVRAISON (numfou, numli);
 alter table DETAILLIVRAISON add constraint fk_detail_livraison_proposer
FOREIGN KEY (numfou, numprod) REFERENCES PROPOSER (numfou, numprod);

INSERT INTO PRODUIT values
 (1, 'Bocal de cornichons'),
 (2, 'Tube de dentifrice'),
 (3, 'Flacon de lotion anti-escarres'),
 (4, 'Déodorant fraîcheur 96 heures');

INSERT INTO FOURNISSEUR values
 (1, 'Bocaux Gérard'),
 (2, 'Paramédical Gisèle'),
 (3, 'Tracteurs Raymond');

INSERT INTO PROPOSER values
 (1, 1, 2),
 (2, 1, 3),
 (2, 2, 2),
 (2, 3, 1);

INSERT INTO LIVRAISON values
 (1, 1, now()),
```

```
 (1, 2, now()),
 (2, 1, now());

INSERT INTO DETAILLIVRAISON values
 (1, 1, 1, 5),
 (1, 2, 1, 2),
 (2, 1, 2, 20),
 (2, 1, 3, 1);
```

.6 Arbre généalogique

La table PERSONNE, le champ pere contient le numéro du père de la personne, le champ mere contient le numéro de la mère de la personne.

```
DROP TABLE IF EXISTS personne;

CREATE TABLE personne
(numpers int PRIMARY KEY,
 nom varchar(30),
 prenom varchar(30),
 pere int,
 mere int,
 FOREIGN KEY (pere) REFERENCES personne(numpers),
 FOREIGN KEY (mere) REFERENCES personne(numpers)
);

insert into personne values (1, 'Estermont', 'Cassana', NULL, NULL);
insert into personne values (2, 'Baratheon', 'Steffon', NULL, NULL);
insert into personne values (3, 'Baratheon', 'Renly', 2, 1);
insert into personne values (4, 'Baratheon', 'Stannis', 2, 1);
insert into personne values (5, 'Baratheon', 'Robert',2, 1);
insert into personne values (12, 'Lannister', 'Joanna', NULL, NULL);
insert into personne values (13, 'Lannister', 'Tywin', NULL, NULL);
insert into personne values (9, 'Lannister', 'Cersei', 13, 12);
insert into personne values (6, NULL, 'Gendry', 5, NULL);
insert into personne values (8, 'Baratheon', 'Tommen', 5, 9);
insert into personne values (7, 'Baratheon', 'Joffrey', 5, 9);
insert into personne values (10, 'Baratheon', 'Myrcella', 5, 9);
insert into personne values (11, 'Lannister', 'Jaime', 13, 12);
insert into personne values (14, 'Lannister', 'Tyrion', 13, 12);
insert into personne values (15, 'Florent', 'Selyse', NULL, NULL);
insert into personne values (16, 'Baratheon', 'Shireen', 4, 15);
```

.7 Comptes bancaires

```
DROP TABLE IF EXISTS OPERATION;
DROP TABLE IF EXISTS TYPEOPERATION;
DROP TABLE IF EXISTS COMPTECLIENT;
DROP TABLE IF EXISTS TYPECCL;
DROP TABLE IF EXISTS PERSONNEL;
DROP TABLE IF EXISTS CLIENT;

CREATE TABLE CLIENT
(numcli int primary key auto_increment,
 nomcli varchar(30),
 prenomcli varchar(30),
```

```sql
 adresse varchar(60),
 tel varchar(10),
 CONSTRAINT ck_telephone CHECK(LENGTH(tel)=10)
);

CREATE TABLE PERSONNEL
(numpers int primary key auto_increment,
 nompers varchar(30),
 prenompers varchar(30),
 manager int,
 salaire int,
 CONSTRAINT ck_salaire CHECK(SALAIRE >= 1254.28)
);

CREATE TABLE TYPECCL
(numtypeccl int primary key auto_increment,
 nomtypeccl varchar(30)
);

CREATE TABLE COMPTECLIENT
(numcli int,
 numccl int,
 numtypeccl int,
 dateccl date not null,
 numpers int,
 CONSTRAINT pk_compteclient
                PRIMARY KEY (numcli, numccl),
 CONSTRAINT fk_ccl_typeccl
                FOREIGN KEY (numtypeccl)
                REFERENCES TYPECCL (numtypeccl),
 CONSTRAINT fk_ccl_client
                FOREIGN KEY (numcli)
                REFERENCES CLIENT (numcli),
 CONSTRAINT fk_ccl_personnel
                FOREIGN KEY (numpers)
                REFERENCES PERSONNEL (numpers)
);

CREATE TABLE TYPEOPERATION
(numtypeoper int primary key auto_increment,
 nomtypeoper varchar(30)
);

CREATE TABLE OPERATION
(numcli int,
 numccl int,
 numoper int,
 numtypeoper int,
 dateoper date not null,
 montantoper int not null,
 libeloper varchar(30),
 CONSTRAINT pk_operation
                PRIMARY KEY (numcli, numccl, numoper),
 CONSTRAINT fk_oper_ccl
                FOREIGN KEY (numcli, numoper)
                REFERENCES COMPTECLIENT (numcli, numccl),
 CONSTRAINT fk_oper_codeoper
                FOREIGN KEY (numtypeoper)
                REFERENCES TYPEOPERATION (numtypeoper),
```

```
 CONSTRAINT montant_operation
                CHECK(montantoper <> 0)
);

INSERT INTO TYPECCL VALUES (1 , 'Compte courant'),
        (2, 'livret'),
        (3, 'PEL');

INSERT INTO TYPEOPERATION VALUES (1, 'dépôt espèces'),
        (2, 'prélèvement'),
        (3, 'virement'),
        (4, 'retrait');
```

.8 Comptes bancaires avec exceptions

```
DROP TABLE IF EXISTS OPERATION;
DROP TABLE IF EXISTS COMPTECLIENT;
DROP TABLE IF EXISTS TYPECCL;
DROP TABLE IF EXISTS TYPEOPERATION;
DROP TABLE IF EXISTS PERSONNEL;
DROP TABLE IF EXISTS CLIENT;

CREATE TABLE CLIENT
(numcli int,
 nomcli varchar(30),
 prenomcli varchar(30),
 adresse varchar(60),
 tel varchar(10)
);

CREATE TABLE PERSONNEL
(numpers int,
 nompers varchar(30),
 prenompers varchar(30),
 manager int,
 salaire int
);

CREATE TABLE TYPECCL
(numtypeccl int,
 nomtypeccl varchar(30)
);

CREATE TABLE COMPTECLIENT
(numcli int,
 numccl int,
 numtypeccl int,
 dateccl date default sysdate not null,
 numpers int
);

CREATE TABLE TYPEOPERATION
(numtypeoper int,
 nomtypeoper varchar(30)
);

CREATE TABLE OPERATION
```

```sql
(numcli int,
 numccl int,
 numoper int,
 numtypeoper int,
 dateoper date,
 montantoper int not null,
 libeloper varchar(30)
);

ALTER TABLE CLIENT ADD
        (
        CONSTRAINT pk_client PRIMARY KEY (numcli),
        CONSTRAINT ck_telephone CHECK(LENGTH(tel)=10)
        );

ALTER TABLE PERSONNEL ADD
        (
        CONSTRAINT pk_personnel PRIMARY KEY (numpers),
        CONSTRAINT ck_salaire CHECK(SALAIRE >= 1254.28)
        );

ALTER TABLE TYPECCL ADD
        CONSTRAINT pk_typeccl PRIMARY KEY (numtypeccl);

ALTER TABLE TYPEOPERATION ADD
        CONSTRAINT pk_typeoperation PRIMARY KEY (numtypeoper);

ALTER TABLE COMPTECLIENT ADD
        (
        CONSTRAINT pk_compteclient
                PRIMARY KEY (numcli, numccl),
        CONSTRAINT fk_ccl_typeccl
                FOREIGN KEY (numtypeccl)
                REFERENCES TYPECCL (numtypeccl),
        CONSTRAINT fk_ccl_client
                FOREIGN KEY (numcli)
                REFERENCES CLIENT (numcli),
        CONSTRAINT fk_ccl_personnel
                FOREIGN KEY (numpers)
                REFERENCES PERSONNEL (numpers)
        );

ALTER TABLE OPERATION ADD
        (
        CONSTRAINT pk_operation
                PRIMARY KEY (numcli, numccl, numoper),
        CONSTRAINT fk_oper_ccl
                FOREIGN KEY (numcli, numoper)
                REFERENCES COMPTECLIENT (numcli, numccl),
        CONSTRAINT fk_oper_codeoper
                FOREIGN KEY (numtypeoper)
                REFERENCES typeoperation (numtypeoper),
        CONSTRAINT montant_operation
                CHECK(montantoper <> 0 AND montantoper >= -1000 AND montantoper <= 1000)
        );

INSERT INTO TYPECCL VALUES (
        (SELECT nvl(MAX(numtypeccl), 0) + 1
        FROM TYPECCL
```

```
        ),
'Compte courant');

INSERT INTO TYPECCL VALUES (
        (SELECT nvl(MAX(numtypeccl), 0) + 1
        FROM TYPECCL
        ),
'livret');

INSERT INTO TYPECCL VALUES (
        (SELECT nvl(MAX(numtypeccl), 0) + 1
        FROM TYPECCL
        ),
'PEL');

INSERT INTO TYPEOPERATION VALUES (
        (SELECT nvl(MAX(numtypeoper), 0) + 1
        FROM TYPEOPERATION
        ),
'dépôt espèces');

INSERT INTO TYPEOPERATION VALUES (
        (SELECT nvl(MAX(numtypeoper), 0) + 1
        FROM TYPEOPERATION
        ),
'prélèvement');

INSERT INTO TYPEOPERATION VALUES (
        (SELECT nvl(MAX(numtypeoper), 0) + 1
        FROM TYPEOPERATION
        ),
'virement');

INSERT INTO TYPEOPERATION VALUES (
        (SELECT nvl(MAX(numtypeoper), 0) + 1
        FROM TYPEOPERATION
        ),
'retrait');
```

.9 Secrétariat pédagogique

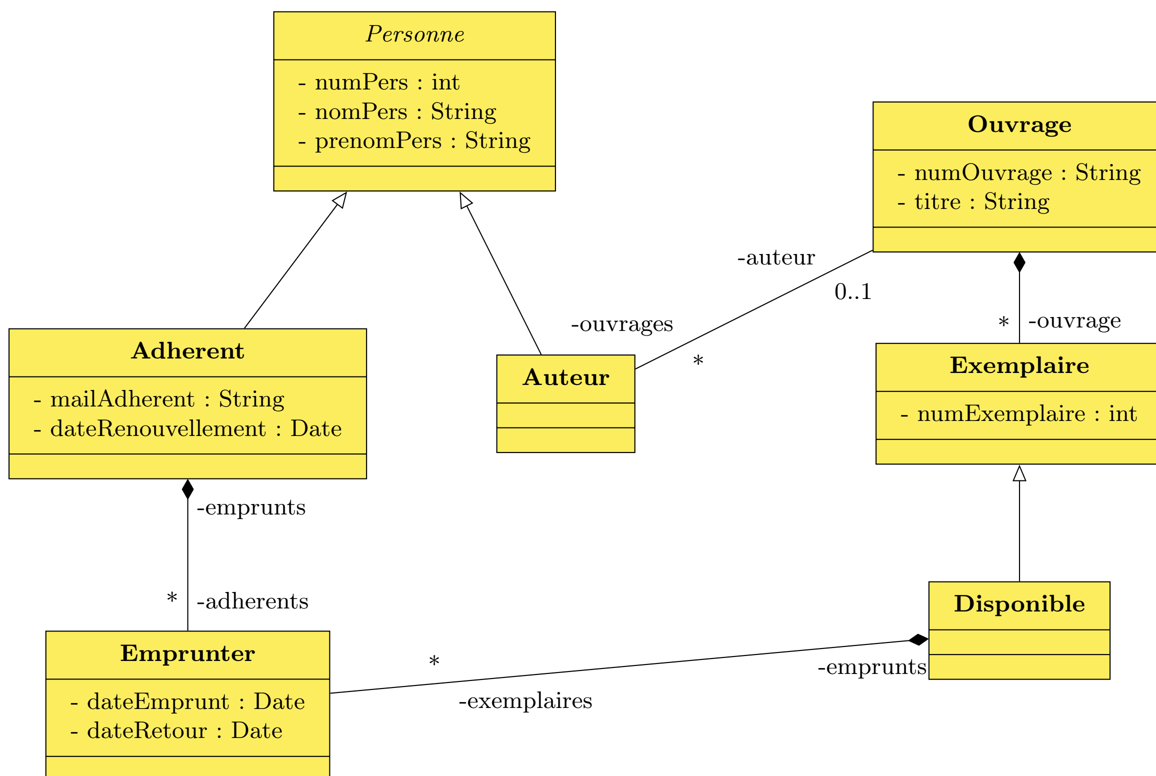
```
DROP TABLE IF EXISTS RESULTAT;
DROP TABLE IF EXISTS EXAMEN;
DROP TABLE IF EXISTS PREREQUIS;
DROP TABLE IF EXISTS INSCRIPTION;
DROP TABLE IF EXISTS MODULE;
DROP TABLE IF EXISTS ETUDIANT;

CREATE TABLE ETUDIANT
        (numEtud int PRIMARY KEY,
        nom varchar(40),
        prenom varchar(40),
        datenaiss date,
        civilite varchar(4),
        patronyme varchar(40),
        numsecu varchar(15) NOT NULL
        );
```

```sql
CREATE TABLE MODULE
        (numMod int PRIMARY KEY,
        nomMod varchar(15),
        effecMax int DEFAULT 30
        );

CREATE TABLE EXAMEN
        (numMod int REFERENCES MODULE(numMod),
        numExam int,
        dateExam date,
        PRIMARY KEY(numMod, numExam)
        );

CREATE TABLE INSCRIPTION
        (numEtud int REFERENCES ETUDIANT(numEtud),
        numMod int REFERENCES MODULE(numMod),
        dateInsc date,
        PRIMARY KEY(numEtud, numMod)
        );

CREATE TABLE PREREQUIS
        (numMod int REFERENCES MODULE(numMod),
        numModPrereq int REFERENCES MODULE(numMod),
        noteMin int NOT NULL DEFAULT 10,
        PRIMARY KEY(numMod, numModPrereq)
        );

CREATE TABLE RESULTAT
        (numMod int,
        numExam int,
        numEtud int,
        note int,
        PRIMARY KEY(numMod, numExam, numEtud),
        FOREIGN KEY (numMod, numExam) REFERENCES EXAMEN(numMod, numExam),
        FOREIGN KEY (numEtud, numMod) REFERENCES INSCRIPTION(numEtud, numMod)
        );

INSERT INTO MODULE (numMod, nomMod) VALUES
(1, 'Oracle'),
(2, 'C++'),
(3, 'C'),
(4, 'Algo'),
(5, 'Merise'),
(6, 'PL/SQL Oracle'),
(7, 'mySQL'),
(8, 'Algo avancee');

INSERT INTO PREREQUIS (numMod, numModPrereq) VALUES
(1, 5),
(2, 3),
(6, 5),
(8, 5),
(7, 5);
INSERT INTO PREREQUIS VALUES (6, 1, 12);
```

.10 Mariages

```sql
CREATE TABLE PERSONNE
(numpers number PRIMARY KEY,
 nom varchar(30) NOT NULL,
 prenom varchar(30),
 pere REFERENCES PERSONNE(numpers),
 mere REFERENCES PERSONNE(numpers)
);

CREATE TABLE MARIAGE
(
nummari NUMBER REFERENCES PERSONNE(numpers),
numfemme NUMBER REFERENCES PERSONNE(numpers),
datemariage DATE DEFAULT SYSDATE,
datedivorce DATE DEFAULT NULL,
PRIMARY KEY(nummari, numfemme, dateMariage)
);
```

.11 Bibliothèque




```sql
drop table emprunter;
drop table exemplaire;
drop table ouvrage;
drop table adherent;
drop table personne;

create table personne
(
numpers int primary key auto_increment,
nompers varchar(64),
```

```sql
prenompers varchar(64)
);

create table adherent
(
numpers int primary key,
mailadherent varchar(64),
daterenouvellement date,
foreign key (numpers) references personne(numpers)
);

create table ouvrage
(
numouvrage int primary key auto_increment,
numauteur int,
titreouvrage varchar(64),
foreign key (numauteur) references personne(numpers)
);

create table exemplaire
(
numouvrage int,
numexemplaire int,
empruntable boolean default true,
primary key (numouvrage, numexemplaire),
foreign key (numouvrage) references ouvrage(numouvrage)
);

create table emprunter
(
numadherent int,
numouvrage int,
numexemplaire int,
dateemprunt date,
dateretour date default null,
primary key (numadherent, numouvrage, numexemplaire, dateemprunt),
foreign key (numadherent) references adherent(numpers),
foreign key (numouvrage, numexemplaire) references exemplaire(numouvrage, numexemplaire),
check (dateemprunt < dateretour)
);

drop view adherents;
create view adherents as
       select p.numpers, nompers, prenompers, mailadherent
       from adherent a, personne p
       where p.numpers = a.numpers;

drop view auteurs;
create view auteurs as
       select *
       from personne
       where numpers not in
             (select numpers
             from adherent
             );

drop view exemplaires;
create view exemplaires as
       select o.numouvrage, numexemplaire, titreouvrage, concat(nompers, ", ", prenompers)
```

```
    as auteur
from personne p, ouvrage o, exemplaire e
where p.numpers = o.numauteur
and o.numouvrage = e.numouvrage;
```