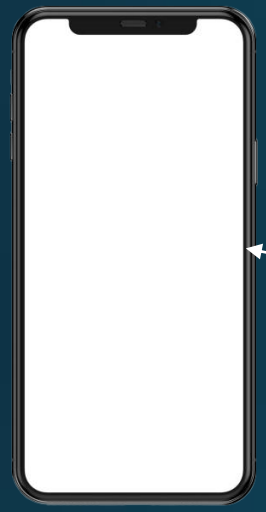
Le SQL


ADRAR
DIGIT@L ACADEMY
PÔLE NUMERIQUE DU CENTRE DE FORMATION ADRAR
> SUPPORT, ADMINISTRATION SYSTEMES & RESEAUX
> DEVELOPPEMENT D'APPLICATIONS WEB & MOBILES
> TRANSFORMATION NUMERIQUE DES ENTREPRISES
http://www.adrar-numerique.com

Le SQL



Le SQL

Pré-requis :
Pour la suite de votre formation et de ce cours, vous allez avoir besoin de certains outils installés sur votre machine.

Installation de Wamp:

- Version x64 pour windows 64 bits : https://sourceforge.net/projects/wampserver/files/latest/download
- Version x86 pour windows 32 bits : https://sourceforge.net/projects/wampserver/files/latest/download

Installation de Worbench:

Commençons par télécharger le logiciel MySQL Community :

- Version x64 pour windows 64 bits : https://dev.mysql.com/downloads/installer/

Définition d'une base données.

SQL veut dire langage de requête structurée. (en anglais : Structured Query Language).

Mais afin d'aller plus loin sur ce cours revenons sur les notions de bases:

- Les bases de données,
- Les SGBDR.

Définition d'une base de données :

Notions de bases:

- Champ
  Un champ est la zone qui permet le stockage des données, correspondant à une colonne dans une vue en liste.
- Enregistrement
  Un enregistrement est un ensemble de valeurs correspondant à une ligne toujours dans une vue en liste.
- Table
  Une table va contenir l'ensemble des enregistrements .
- Index
  Un index est le conteneur des clés.

Le SQL

id	nom	prénom	profession	code postal	ville
1	Durand	Michel	Directeur	75016	Paris
2	Dupond	Karine	Secrétaire	92000	Courbevoie
3	Mensoif	Gérard	Commercial	75001	Paris
4	Monauto	Alphonse	Commercial	75002	Paris
5	Emarre	Jean	Employé	75015	Paris
6	Abois	Nicole	Secrétaire	95000	St Denis
7	Dupond	Antoine	Assistant commercial	75014	Paris

intitulé du champ

enregistrement

Champ

Table

Le SQL

- Une base de données est un ensemble qui permet le stockage des données.
- Les données sont écrites de manière structurées ce qui signifie que chaque donnée est enregistrée dans un champ qui est inclus dans une table.
- Lorsque les tables ont des relations entre elles, on parle alors de bases de données relationnelles.
- Ces tables peuvent être indexées pour permettre des accès plus rapides à certaines données et permettre aussi des tris.

Le SGBR ou SGBDR

Le SGBDR va gérer l'accès à la base de données. Aucun logiciel n'accédera directement à la base de données, seul le SGBD le fera.

Cela implique que le SGBDR :

- disposera d'un langage pour pouvoir dialoguer avec les applications, (notamment le SQL).
- gérera l'écriture et la lecture des données,
- mettra à disposition les tables, et les droits associés,
- gérera le partage des données,
- s'assurera de l'intégrité des données,
- gérera la relation entre les tables (dans le cas de base de données relationnelles)

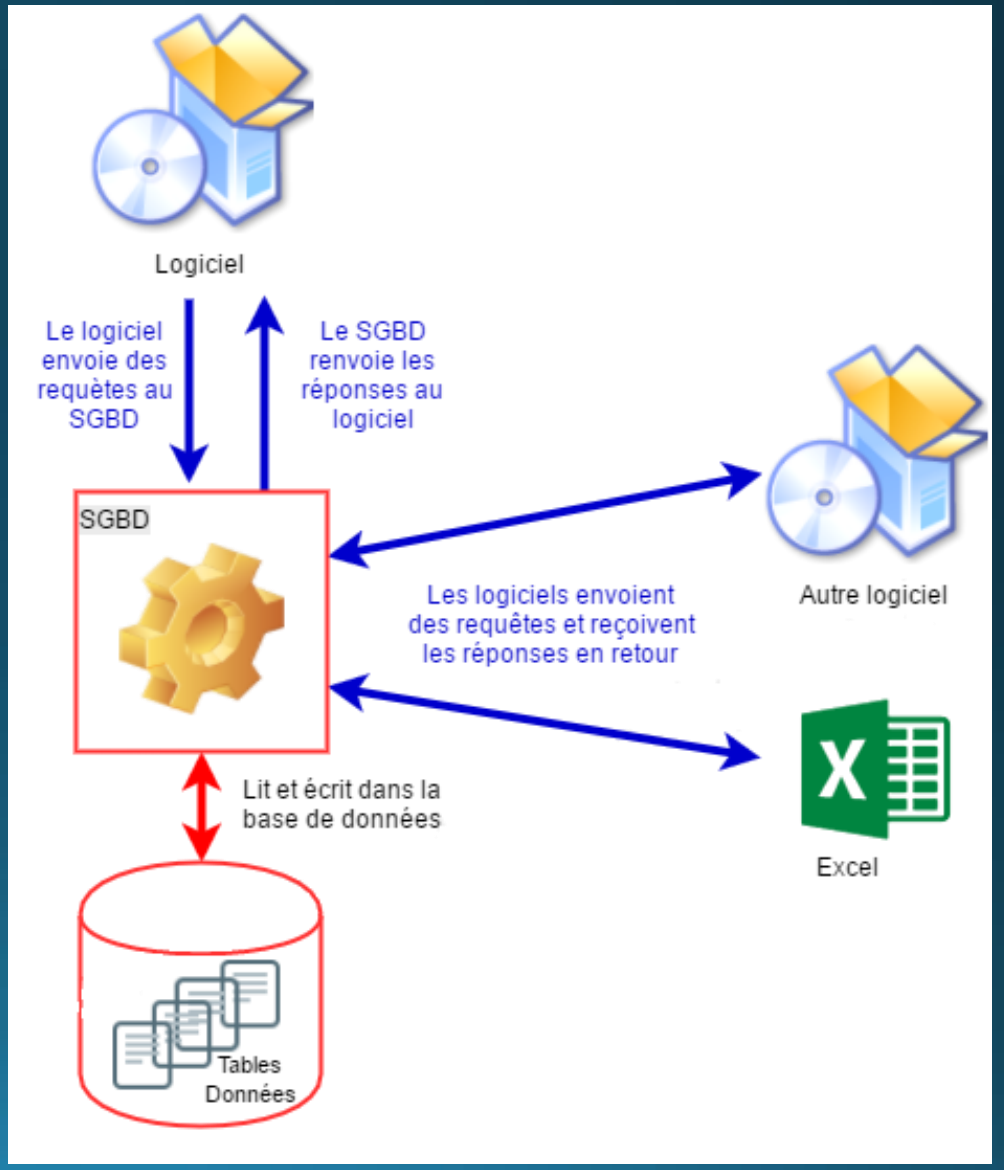
Le SQL

Les avantages à utiliser les bases de données sont les suivants :

Une standardisation des accès :
Chaque logiciel accède aux données via le SGBD en utilisant un langage standardisé. Il est possible de changer de base de données sans avoir un réécrire le logiciel.

Un partage des données ainsi que l'accès simultané :
Le SGBD gère les accès à la base de données, donc il gère aussi le partage au données. Il est donc possible d'avoir plusieurs logiciels qui lisent et écrivent en même temps sur la base de données.

Une grande fiabilité des données :
Chaque logiciel ne peut pas faire n'importe quoi. C'est le SGBD qui gère l'enregistrement des données.



Les principaux SGBDR utilisés en entreprise sont:

- Oracle : sans doute le plus connu,
- MySQL : Un système gratuit issu du monde libre,
- SQLServer : Le SGBD de Microsoft.

Bon à savoir : vous pourrez trouver les appellations de base de données sous les formes suivantes :

- BdD: Base de Données
- BD: Base Données
- DB: DataBase (nom en anglais)

Le SQL

Afin que les logiciels utilisés et le SGBD puissent se comprendre, ils utilisent un langage appelé SQL. Ce langage complet va être utilisé pour:

- Lire les données,
- Ecrire les données,
- Modifier les données,
- Supprimer les données

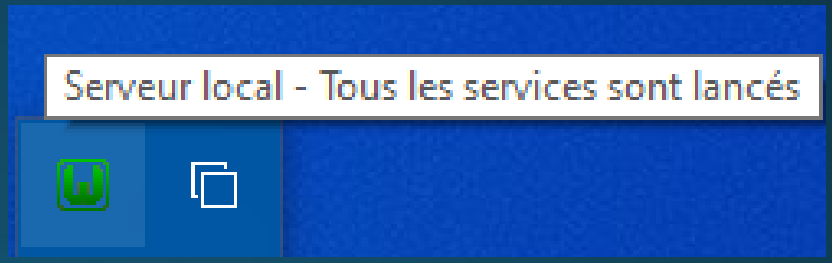
Il permettra aussi de modifier la structure de la base de données :

- Ajouter des tables,
- Modifier les tables,
- les supprimer
- Ajouter, ou supprimer des utilisateurs,
- Gérer les droits des utilisateurs,
- Gérer les bases de données : en créer de nouvelles, les modifier, etc …

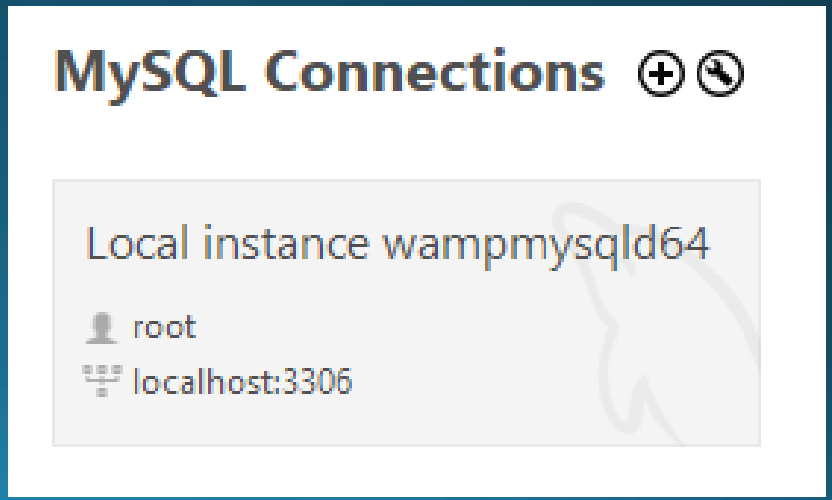
Démarrer son serveur local:

Lancez votre serveur wamp :

Vérifiez que votre serveur est bien lancé :



Lancez maintenant MySqlWorkbench puis cliquez sur votre instance locale :



Les requêtes sur la structure.

La requête **CREATE:**

Cette requête servira principalement à créer vos tables dans votre BDD, mais également à créer votre base de données. Mais étudions ensemble l'exemple ci-contre:

```
CREATE DATABASE test;
use test;
CREATE TABLE `users` (
  idUser bigint(20) NOT NULL,
  name varchar(50) NOT NULL,
  password varchar(255) NOT NULL,
  mail varchar(100) NOT NULL,
  age int(20) DEFAULT NULL,
  idTown bigint(20) DEFAULT NULL
) ;
```

La ligne « create database test » :

Sur cette ligne, nous voyons bien notre requête ‘create’, ensuite le terme ‘database’ est le mot anglais signifiant base de données et le mot ‘test’ indique le nom que nous souhaitons donner à cette BDD et se termine par ‘ ; ’ pour signaler la fin de notre requête. Nous pouvons donc traduire cette requête ainsi = ‘créer base de données test’.

La ligne « use » :

elle indique à notre SGBD que nous souhaitons utiliser la BDD ‘test’ précédemment créé (dans le cas où celui-ci gère plusieurs bases de données). Elle se termine par ‘ ; ’ pour signaler la fin de notre requête.

```sql
CREATE DATABASE test;
use test;
CREATE TABLE `users` (
  idUser bigint(20) NOT NULL,
  name varchar(50) NOT NULL,
  password varchar(255) NOT NULL,
  mail varchar(100) NOT NULL,
  age int(20) DEFAULT NULL,
  idTown bigint(20) DEFAULT NULL
) ;
```

Nous arrivons maintenant à la dernière ligne.
Effectivement, vous pouvez constater que cette requête comporte plusieurs éléments séparés par des retours à la ligne afin des les différencier.

La ligne « CREATE TABLE 'users'» :

Sur cette ligne nous voyons bien notre requête 'CREATE', ensuite le terme 'TABLE' indique ce que nous voulons créer et le mot 'users' indique le nom que nous souhaitons donner à cette table et se termine par ' ; ' pour signaler la fin de notre requête.

```
CREATE DATABASE test;
use test;
CREATE TABLE `users` (
  idUser bigint(20) NOT NULL,
  name varchar(50) NOT NULL,
  password varchar(255) NOT NULL,
  mail varchar(100) NOT NULL,
  age int(20) DEFAULT NULL,
  idTown bigint(20) DEFAULT NULL
) ;
```

‘idUser’ sera le nom du 1er champ de notre table, et à partir de là nous pouvons donc en conclure que ‘name’, ‘password’, ‘mail’, ‘age’, ‘idtown’ seront les noms de nos autres champs.
Cependant, le champ ‘idtown’ correspond également à la clé primaire d’une autre table. Nous l’aborderons donc à la suite de ce cours dans la partie ‘’requête alter table’’.

‘bigint’, ‘varchar’, ‘int’ correspondent au type du champ il peut être numérique (int , double, real, float, boolean), une chaine de caractère (char, varchar, text, blob, enum, set), une date(date, datetime, timestamp), spatial (polygon, point, geometry) ou un JSON.

```sql
CREATE DATABASE test;
use test;
CREATE TABLE `users` (
  idUser bigint(20) NOT NULL,
  name varchar(50) NOT NULL,
  password varchar(255) NOT NULL,
  mail varchar(100) NOT NULL,
  age int(20) DEFAULT NULL,
  idTown bigint(20) DEFAULT NULL
) ;
```

(20), (50), (100), etc … indiquent la taille ou la valeur maximale du champ (non obligatoire pour certains types, comme le 'text' par exemple).

'NOT NULL', 'DEFAULT NULL', signalent à notre SGBD les valeurs par défaut de nos divers champs lors d'un enregistrement. Cela n'est pas obligatoire on aurait très bien pu écrire « password varchar(255), » sans conséquences pour notre requête. Cependant, si nous indiquons NOT NULL, le SGBD attendra de recevoir une valeur lors de la demande d'insertion d'un nouvel enregistrement, faute de quoi il nous retournera une erreur. Au contraire, Le DEFAULT NULL demande de remplir un champ avec une valeur null au cas ou il ne reçoit pas de valeur pour ce champ.

```
CREATE DATABASE test;
use test;
CREATE TABLE `users` (
  idUser bigint(20) NOT NULL,
  name varchar(50) NOT NULL,
  password varchar(255) NOT NULL,
  mail varchar(100) NOT NULL,
  age int(20) DEFAULT NULL,
  idTown bigint(20) DEFAULT NULL
) ;
```

La requête ALTER TABLE:

Cette requête vous permettra de modifier les tables dans votre BDD.
Mais afin de voir cela ensemble, vous aller reprendre la partie précédente en créant votre BDD 'test' sur votre propre serveur, ainsi que la table 'users' et la table 'towns' ayant les champs :

- idTown dont le type est un entier,
- townName dont le type est un varchar,
- townCp dont le type est un varchar.

```
CREATE TABLE towns (
   idTown bigint(20),
   townname varchar(100),
   towncp varchar(50)
);
```

Maintenant que nos deux tables sont créées, nous allons pouvoir les modifier.

```
ALTER TABLE users
   ADD PRIMARY KEY (idUser),
   ADD KEY idTown (idTown);
```

Cette requête demande à notre SGBD de modifier notre table « users ».
Lors de celle-ci on lui indique d'ajouter des clés (KEY) qui serviront d'index grâce au terme 'ADD' .
PRIMARY KEY (idUser) signifie clé primaire (PK). Cette dernière est unique et est attribuée au champ idUser de la table. Elle nous permettra d'utiliser l'enregistrement lui correspondant.
KEY idTown (idTown) est une clé index. Elle provient du champ idTown de la table (towns) et sera associée au champ idTown de la table users.

```
ALTER TABLE users
   ADD CONSTRAINT FOREIGN KEY (idtown) REFERENCES towns (idtown);
```

Ici, nous ajoutons une contrainte sur notre table ‘users ’. En effet, nous demandons à notre SGBD d’ajouter une contrainte avec le terme ADD CONSTRAINT .
Cette contrainte sera une FOREIGN KEY (FK) ‘ou clé étrangère en français’.
Cette FK correspondra au champ ‘idTown’ de la table ‘users’ et, comme nous l’indique REFERENCES, elle fera donc références à la table ‘towns’ et précisément à son champ ‘idTown’.

```
ALTER TABLE users
   MODIFY iduser bigint(20) NOT NULL AUTO_INCREMENT;
```

Sur cette requête nous voulons modifier ‘MODIFY’ notre champ idUser, qui est également une PK, en indiquant à notre SGBD que ce champ ne peut pas être null ‘NOT NULL’, mais également qu’il doit s’auto-incrémenter ‘AUTO_INCREMENT’ à chaque nouvel enregistrement.

Bon à savoir : pour renommer une table la requête est la suivante = « RENAME TABLE ‘nom_à_changer’ TO ‘nouveau_nom’ ».

Le SQL

```
ALTER TABLE users
    add column phone varchar(50);
```

Cette fois nous demandons d'ajouter une colonne (ou champ) 'add column' qui sera nommé 'phone' et de type varchar avec une taille de 50.

Je m'aperçois que le nom ou le type de la colonne ne vont pas convenir donc je vais les modifier :

```
ALTER TABLE users
    change phone telephone bigint(100);
```

Dans cet exemple j'ai changé le nom de la colonne et son type. Pour y parvenir vous constaterez que j'indique d'abord le nom à changé puis le nom souhaité et enfin le type du champ.

```
ALTER TABLE users
    drop phone ;
```

```
ALTER TABLE users
    drop column phone;
```

Ces 2 exemples font la même chose. Ils servent à supprimer une colonne d'une table.

```
ALTER TABLE users
    modify telephone varchar(100);
```

Ici, nous modifions notre colonne notamment son type. Nous pouvons garder le même type et changer sa taille également.

La requête DROP TABLE :

Cette requête est assez simple à écrire et à comprendre. En effet elle sert à supprimer une table de notre BDD.

```
drop TABLE roleBean;
```

Le SQL

Pour conclure sur cette partie, je vous invite d'abord de mettre à jour votre BDD en effectuant les requêtes nécessaires.
Et voici un exemple de création de table reprenant ce que nous venons de voir en condensant un peu l'écriture.

```
CREATE TABLE users (
  idUser bigint(20) NOT NULL primary key auto_increment,
  idsession varchar(255),
  name varchar(50),
  password varchar(255),
  mail varchar(100),
  phone varchar(50),
  idtown bigint(20),
  FOREIGN KEY (idtown) REFERENCES townBean (idtown)
);
```

Maintenant petit exercice

Dans une Database ‘facturation’, j’ai trois tables :

- Clients
- Factures
- Villes

La table clients aura 5 champs dont :

- idClient
- nom
- prenom
- telephone
- idVille

La table factures comportera 3 champs :

- idFacture
- numeroFacture
- idClient

La table villes intégrera 3 champs :

- idVille
- nom
- codePostal

Le SQL

```sql
create database facturation;
use facturation;
CREATE TABLE villes(
   idVille INT PRIMARY KEY auto_increment,
   nom VARCHAR(100),
   codePostal VARCHAR(50)
);

CREATE TABLE clients(
   idClient INT PRIMARY KEY auto_increment,
   nom VARCHAR(50),
   prenom VARCHAR(50),
   telephone VARCHAR(50),
   idVille INT,
   FOREIGN KEY(idVille) REFERENCES villes(idVille)
);

CREATE TABLE factures(
   idFacture INT PRIMARY KEY auto_increment,
   numeroFacture VARCHAR(50),
   idClient INT,
   FOREIGN KEY(idClient) REFERENCES clients(idClient)
);
```

Toutefois nous avons étudié des requêtes sur des tables avec lesquelles nous avions que des relations 0,1 ; 0,n.

Pour expliquer la relation entre la table clients et la table villes, nous pourrions dire qu'un client peut habité 0 ville si celle-ci n'est pas enregistrée dans la BDD jusqu'à 1 seule ville.

A l'inverse, une ville peut-être habitée par aucun client jusqu'à 'n' clients, s'il y a plusieurs clients qui vivent dans la même ville.

Il faut aussi savoir que des tables sont appelées 'table d'association' car leur clé primaire correspond aux clés primaires de deux autres tables lors d'une relation 0,n ; 0,n entre ces 2 tables.

Le SQL

Dans cet exemple nous voyons la table ville avec 2 champs 'id_ville' et 'nom', ainsi que la table cp (code postal) également avec 2 champs 'id_cp' et 'cp'.

En reprenant la logique vue juste avant, nous pouvons ainsi dire qu'une ville peut avoir entre 0 et 'n' codes postaux et un code postal peut avoir entre 0 et 'n' villes

Cela se traduira donc par la création de la table d'association correspondre dont la clé primaire sera formée avec les clés étrangères id_ville et id_cp des tables ville et cp.

```sql
CREATE TABLE correspondre(
   id_ville INT,
   id_cp INT,
   PRIMARY KEY(id_ville, id_cp),
   FOREIGN KEY(id_ville) REFERENCES ville_1(id_ville),
   FOREIGN KEY(id_cp) REFERENCES cp_1(id_cp)
);
CREATE TABLE ville(
   id_ville INT primary key auto_increment,
   nom VARCHAR(50)
);

CREATE TABLE cp(
   id_cp INT primary key auto_increment,
   cp VARCHAR(50)
);
```

Les requêtes sur les enregistrements

La requête insert into

Cette méthode nous permettra d'insérer des données dans une base en créant un nouvel enregistrement .

Elle a 2 syntaxes principales.

1. Insérer une ligne en indiquant les informations pour chaque colonne existante (en respectant l'ordre),

   INSERT INTO table VALUES ('valeur 1', 'valeur 2', ...)

   et avec celle-ci nous avons les avantages et inconvénients suivants =

   - Obliger de remplir toutes les données, tout en respectant l'ordre des colonnes
   - Il n'y a pas le nom de colonne, donc les fautes de frappe sont limitées. Par ailleurs, les colonnes peuvent être renommées sans avoir à changer la requête
   - L'ordre des colonnes doit resté identique sinon certaines valeurs prennent le risque d'être complétée dans la mauvaise colonne

2. Insérer une ligne en spécifiant les colonnes que vous souhaiter compléter. Il est possible d'insérer une ligne renseignant seulement une partie des colonnes,

   INSERT INTO table (nom_colonne_1, nom_colonne_2, ...) VALUES ('valeur 1', 'valeur 2', ...)

   et avec elle, il est possible =

   - De ne pas renseigner toutes les colonnes.
   - L'ordre des colonnes n'est pas important.

Il est également possible d'insérer plusieurs enregistrements en 1 seule requête dont voici un exemple :

```
INSERT INTO users (password, name, idTown) VALUES
("pass1", "test1", 33330),
("pass2", "test2", NULL),
("pass3", "test3", NULL),
("pass4", "test4", NULL);
```

La requête UPDATE

Cette méthode nous autorisera à effectuer des modifications dans une base sur des enregistrements déjà existants.

Souvent, cette commande est utilisé avec WHERE, qui est une contrainte permettant d'extraire des enregistrements d'une table si ceux-ci respectent la condition.

La syntaxe de base est écrite ainsi :

```
UPDATE table
SET nom_colonne_1 = 'nouvelle valeur'
WHERE condition
```

Si la condition WHERE n'est pas utilisée, toutes les données du champ nom_colonne_1 serait alors modifiées.

Le SQL

```
UPDATE users
SET idTown = "11"
WHERE idUser = 3;
```

idUser	idsession	name	password	mail	phone	idtown
1	NULL	test1	pass1	NULL	NULL	33330
2	NULL	test2	pass2	NULL	NULL	NULL
3	NULL	test3	pass3	NULL	NULL	11
4	NULL	test4	pass4	NULL	NULL	NULL

```
UPDATE users
SET idTown = "74";
```

idUser	idsession	name	password	mail	phone	idtown
1	NULL	test1	pass1	NULL	NULL	74
2	NULL	test2	pass2	NULL	NULL	74
3	NULL	test3	pass3	NULL	NULL	74
4	NULL	test4	pass4	NULL	NULL	74

La requête DELETE

Cette commande permet de supprimer des enregistrements dans une table.

En associant cette commande avec WHERE, il est donc possible de sélectionner seulement les lignes soumises à la condition.

Bon à savoir: *Avant de lancer une requête delete, il est fortement recommandé de faire une sauvegarde de la base de données, ou à minima de la table concernée, afin de permettre une récupération en cas de mauvaise manipulation.*

La syntaxe est la suivante :

```
DELETE FROM `table`
WHERE condition
```

Attention : en l'absence de la condition WHERE toutes les lignes seront supprimées ce qui videra votre table..

Reprenons maintenant notre exemple pour voir cette commande de plus près.
En ce moment notre table contient les valeurs suivantes :

idUser	idsession	name	password	mail	phone	idtown
1	*NULL*	test1	pass1	*NULL*	*NULL*	74
2	*NULL*	test2	pass2	*NULL*	*NULL*	74
3	*NULL*	test3	pass3	*NULL*	*NULL*	74
4	*NULL*	test4	pass4	*NULL*	*NULL*	74

```
DELETE FROM users
WHERE idUser = 2;
```

idUser	idsession	name	password	mail	phone	idtown
1	NULL	test1	pass1	NULL	NULL	74
3	NULL	test3	pass3	NULL	NULL	74
4	NULL	test4	pass4	NULL	NULL	74

Nous avons bien supprimer l'enregistrement (ou ligne) dont le champ idUser était égale à 2.

```
DELETE FROM users
WHERE idUser < 3;
```

idUser	idsession	name	password	mail	phone	idtown
3	NULL	test3	pass3	NULL	NULL	74
4	NULL	test4	pass4	NULL	NULL	74

Cette requête demandait que tous les enregistrements où le champ idUser était inférieur à 3, soient supprimés.

Enfin, nous avons abordé le fait qu'en l'absence de condition (where),

```
DELETE FROM users
```

toutes les données de la table seront supprimées de celle-ci.
Cette commande pourra vous être effectivement utile mais cependant, un des inconvénients possible selon vos besoins, est que cette méthode ne réinitialise pas l'auto-incrémentation.
Cela signifie que si votre id de votre dernier enregistrement était par exemple de 185, le prochain enregistrement reprendra à 186 .

Toutefois, la commande TRUNCATE vous permettra de palier a ceci avec la syntaxe =

```
TRUNCATE TABLE `utilisateur`
```

qui aura l'avantage de réinitialiser l'auto-incrémentation et vous permettre de repartir de 0.

Exercice 1ère partie.

En tant que gérant d'une société nationale de location de voitures, j'ai sous ma responsabilité :
des responsables d'agences,
des locaux situés dans diverses villes,
une flotte de voitures.

Les responsables d'agences ont pour attributs leurs noms, prénoms, téléphones et mails et ville de l'agence sous sa responsabilité.
Les locaux ont pour caractéristiques une adresse avec rue, ville et code postal.
Les voitures peuvent être louées ou non , elles ont également une marque, une couleur et peuvent être prises et rendues dans n'importe laquelle de mes agences.

J'aurais besoin de sauvegarder mes données et les modifier en temps réel.

Exercice 2ème partie:

Je viens de faire l'acquisition d'une nouvelle agence situe avenue bloblocar, 22 222, Toukilou. Son responsable d'agence sera FourWheels Joulou joignable au 88.11.55.11.88 et toukilou@atoutroule.fr .

J'ai acheté 2 nouveaux véhicules une fiot 7014 jaune, et une farp kagi orange.

La 3ème des voitures créés en première partie d'exercice a eu un accident et nous avons dû changé sa couleur en marron clair.

Le nom du champ 'marque' de la table 'voiture' ne correspond pas à notre réalité, renommer la colonne en 'marque_et_modele'.

Nous nous sommes aperçu que la colonne 'mail' est trop petite pour nos futurs emails, modifiez la taille du champ 'mail'.

Le champ 'louer_ou_pas' n'est pas nécessaire pour nos besoins, supprimez cette colonne.
La seconde voiture créé dans la première partie ne roulera plus car elle est partie à la casse, retirez la de la BDD.

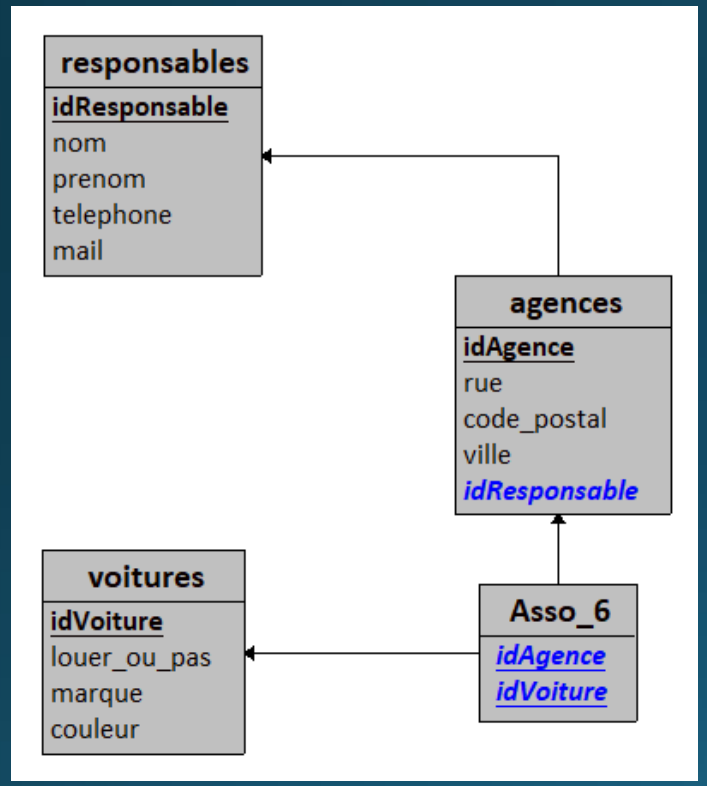
Pour les plus rapides:

Le responsable FourWheels Joulou a changé de telephone qui est maintenant ‘46.25.63.14.98’ , modifiez son numéro avec une condition sur le nom et prenom.

Ajoutez une colonne modele dans la table ‘voitures’, puis renommez la colonne marque_et_modele en marque.

Modifiez la 1ère voiture en changeant sa couleur en ‘marron clair’ puis supprimez toutes les voitures qui sont marrons claires.

Le SQL




```sql
create database exo;
use exo;
CREATE TABLE responsables(
   idResponsable INT PRIMARY KEY auto_increment,
   nom VARCHAR(50),
   prenom VARCHAR(50),
   telephone VARCHAR(50),
   mail VARCHAR(100)
);
CREATE TABLE agences(
   idAgence INT PRIMARY KEY auto_increment,
   rue VARCHAR(200),
   code_postal VARCHAR(100),
   ville VARCHAR(100),
   idResponsable INT,
   FOREIGN KEY(idResponsable) REFERENCES responsables(idResponsable)
);
CREATE TABLE voitures(
   idVoiture INT PRIMARY KEY auto_increment,
   louer_ou_pas smallint,
   marque VARCHAR(50),
   couleur VARCHAR(50)
);
CREATE TABLE abriter(
   idAgence INT,
   idVoiture INT,
   PRIMARY KEY(idAgence, idVoiture),
   FOREIGN KEY(idAgence) REFERENCES agences(idAgence),
   FOREIGN KEY(idVoiture) REFERENCES voitures(idVoiture)
);
```

Le SQL

```
insert into voitures (louer_ou_pas, marque, couleur) values
("0", "voit1", "rouge"),
("0", "voit2", "blanche"),
("1", "voit3", "bleu"),
("0", "voit4", "noire");
insert into responsables (nom, prenom, telephone, mail) values
("Radial", "Paul", "77.88.99.88.14", "jroule@atoutroule.fr"),
("Royaluni", "Henri", "25.63.41.69.49", "tataouine@atoutroule.fr"),
("Michemiche", "Pierre", "34.64.21.64.78", "losbagnos@atoutroule.fr"),
("Conti", "Arthur", "98.76.15.34.15", "etailleurs@atoutroule.fr");
insert into agences (rue, code_postal, ville) values
("rue Atouteblinde", "36521", "Trop-de-la-balle"),
("avenue Vive-allure", "62541", "Marreteplus"),
("impasse limitation-de-vitesse", "44551", "Mincealors"),
("boulevard Grosexcès", "77889", "Jen-peu-plus");
```

```
insert into agences (rue, code_postal, ville) values
("avenue bloblocar", "22222", "Toukilou");
```

```
insert into responsables (nom, prenom, telephone, mail) values
("FourWheels", "Joulou", "64.15.42.19.76", "toukilou@atoutroule.fr");
```

```
insert into voitures (marque, couleur) values
("fiot 7014", "jaune"),
("farp kagi", "orange");
```

Le SQL

```sql
update voitures
set couleur = "marron clair"
where idVoiture = 3;
```

```sql
update responsables
set telephone = "46.25.63.14.98"
where nom = "FourWheels"
and prenom = "Joulou";
```

```sql
alter table voitures
change marque marque_et_modele varchar(100);
```

```sql
alter table voitures
add column modele varchar(50);
```

```sql
alter table responsables
modify mail varchar(255);
```

```sql
alter table voitures
change marque_et_modele marque varchar(50);
```

```sql
alter table voitures
drop louer_ou_pas;
```

```sql
update voitures
set couleur = "marron clair"
where idVoiture = 1;
```

```sql
delete from voitures
where idVoiture = 2;
```

```sql
delete from voitures
where couleur = "marron clair";
```

Les requêtes de consultations

La commande SELECT :

La commande SELECT permet de faire des opérations de recherche et de calcul à partir des enregistrements d'un ou plusieurs attributs d'une table:

```
SELECT nom_attribut
FROM table1 ;
```

Et pour sélectionner plusieurs attributs on les sépare par des virgules:

```
SELECT nom_attribut1, nom_attribut2
FROM table1 ;
```

Et pour sélectionner tous les attributs :

```
SELECT *
FROM table1 ;
```

La commande DISTINCT :

La commande SELECT peut potentiellement afficher des enregistrements en double.
La commande DISTINCT permet d'éviter des redondances dans les résultats.

```
SELECT DISTINCT nom_attribut
FROM table1 ;
```

La commande AS :

La commande AS permet d'utiliser des alias pour renommer temporairement un attribut ou une table dans une requête.
Cela permet de facilité la lecture des requêtes.
A l'exécution la colonne prendra le nom de l'alias.

```
SELECT nom_attribut1 AS na1, nom_attribut2 AS na2
FROM table1 ;
```

Ou aussi

```
SELECT nom_attribut1 na1, nom_attribut2 na2
FROM table1 ;
```

Pour une table

```
SELECT nom_attribut1
FROM table1 AS t1 ;
```

La commande WHERE :

La commande WHERE permet d'extraire des enregistrements d'une base de données qui respectent une condition.
Cela permet d'obtenir uniquement les informations désirées.

```
SELECT nom_attribut1, nom_attribut2
FROM table1
WHERE condition;
```

Les opérateurs de comparaison qui peuvent être utilisés avec le WHERE :

Opérateur	Description
=	Egale
<>	Pas égale
!=	Pas égale
>	Supérieur à
<	Inférieur à
>=	Supérieur ou égal à
<=	Inférieur ou égal à
IN	Liste de plusieurs valeurs possibles
BETWEEN .... AND	Valeur comprise dans un intervalle de données
LIKE	Recherche en spécifiant le début, le milieu ou la fin d'un mot
IS NULL	Valeur est nulle
IS NOT NULL	Valeur n'est pas nulle

Les opérateurs logiques And/Or :

Les opérateurs logiques AND et OR peuvent être utilisés dans la commande WHERE pour combiner des conditions.

- L'opérateur And dont la syntaxe est :

```
SELECT nom_attribut1, nom_attribut2
FROM table1
WHERE condition1 AND condition2;
```

permet d'afficher un résultat seulement si les conditions 1 et 2 sont vérifiées.

- L'opérateur Or dont la syntaxe est :

```
SELECT nom_attribut1, nom_attribut2
FROM table1
WHERE condition1 OR condition2;
```

Permet d'afficher un résultat dès lors qu'une des 2 conditions est vérifiée.

Il est également possible de combiner ces 2 opérateurs logiques ce qui pourrait donner une syntaxe comme celle-ci par exemple:

```
SELECT nom_attribut1, nom_attribut2
FROM table1
WHERE condition1 AND (condition2 OR condition3);
```

La commande Order By:

La commande ORDER BY permet de trier des lignes dans un résultat d'une requête.
Il est possible de trier les données sur un ou plusieurs attributs par ordre ascendant ou descendant.

la syntaxe est :

```
SELECT nom_attribut1, nom_attribut2
FROM table1
ORDER BY nom_attribut1
```

Par défaut, les résultats sont classés par ordre ascendant.
Pour trier le résultat par ordre décroissant nous utilisons le suffixe DESC :

```
SELECT nom_attribut1, nom_attribut2
FROM table1
ORDER BY nom_attribut1 DESC, nom_attribut2 ASC ;
```

La commande Group By:

La commande GROUP BY est utilisée avec les fonctions de calcul.
Elle permet de regrouper le résultat.
Elle est obligatoire sur tous les attributs qui entourent la fonction de calcul dans le SELECT.

```
SELECT nom_attribut1, fonction (nom_attribut2)
FROM table1
GROUP BY nom_attribut1;
```

Cette commande doit toujours s'utiliser après la commande WHERE et avant la commande HAVING.
Les fonctions possibles :

Fonctions	Description
AVG()	Calcul de la moyenne d'un ensemble de valeurs
COUNT()	Pour compter le nombre de lignes concernées
MAX()	Permet de récupérer la plus haute valeur
MIN()	Permet de récupérer la plus petite valeur
SUM()	Somme de plusieurs lignes

La commande Having:

Cette commande est presque similaire au WHERE à la seule différence que le HAVING permet de filtrer en utilisant les fonctions de calcul.

```
SELECT nom_attribut1, fonction (nom_attribut2)
FROM table1
GROUP BY nom_attribut1
HAVING fonction (nom_attribut2) = condition1;
```

Les Jointures sur les requêtes de consultation

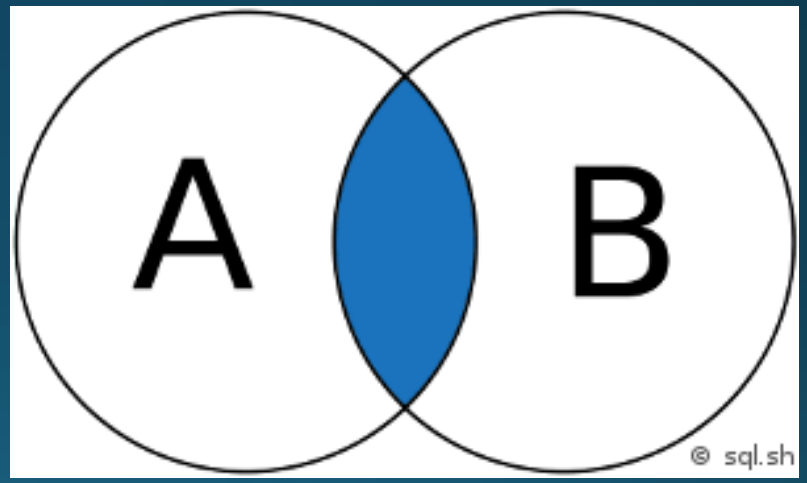
Les jointures permettent d'associer plusieurs tables dans une requête en utilisant la clé étrangère.

Il existe plusieurs méthodes pour associer 2 tables ensemble:

1. INNER JOIN
2. CROSS JOIN
3. LEFT JOIN (ou LEFT OUTER JOIN)
4. RIGHT JOIN (ou RIGHT OUTER JOIN)
5. FULL JOIN (ou FULL OUTER JOIN)
6. SELF JOIN
7. NATURAL JOIN

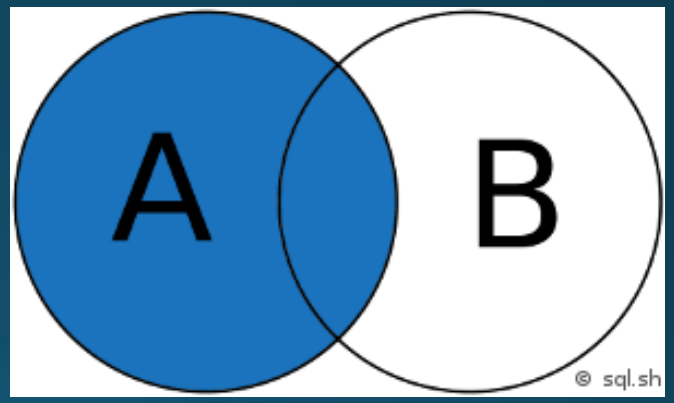
INNER JOIN : jointure interne pour retourner les enregistrements quand la condition est vrai dans les 2 tables. C'est l'une des jointures les plus communes.

```
SELECT *
FROM A
INNER JOIN B ON A.key = B.key
```



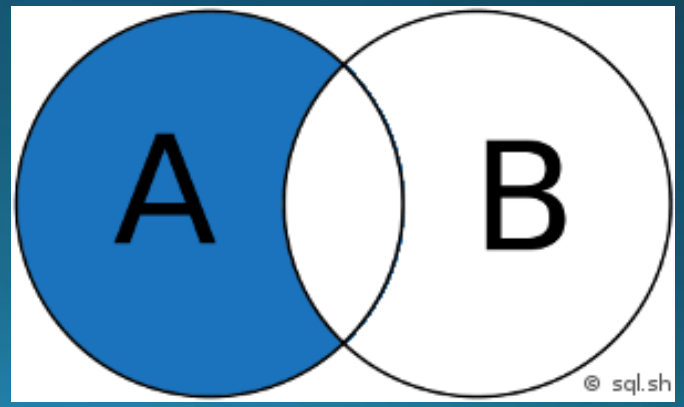
LEFT JOIN : jointure externe pour retourner tous les enregistrements de la table de gauche (LEFT = gauche) même si la condition n'est pas vérifié dans l'autre table.

```
SELECT *
FROM A
RIGHT JOIN B ON A.key = B.key
```



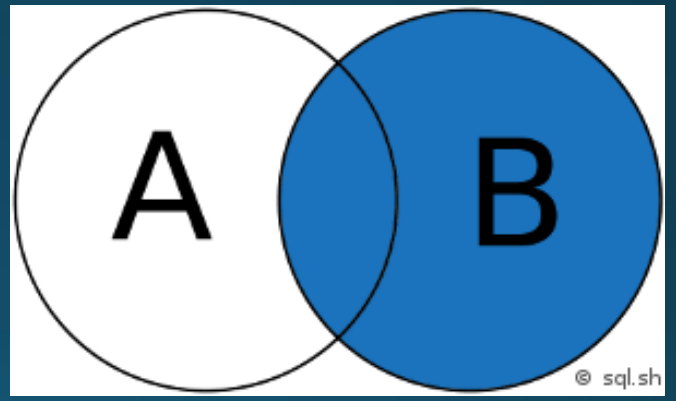
(sans l'intersection de B)

```
SELECT *
FROM A
RIGHT JOIN B ON A.key = B.key
WHERE B.key IS NULL
```

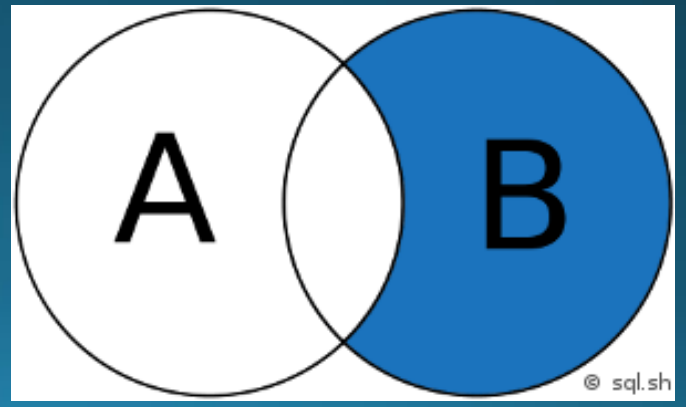


RIGHT JOIN : jointure externe pour retourner tous les enregistrements de la table de droite (RIGHT = droite) même si la condition n'est pas vérifié dans l'autre table.

```
SELECT *
FROM A
RIGHT JOIN B ON A.key = B.key
```

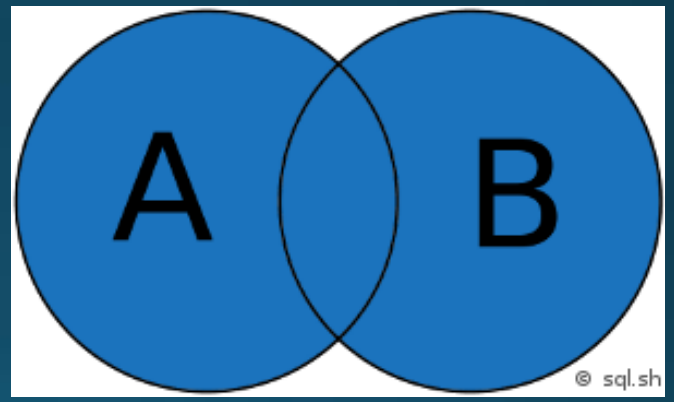



```
(sans l'intersection de A)
SELECT *
FROM A
RIGHT JOIN B ON A.key = B.key
WHERE B.key IS NULL
```



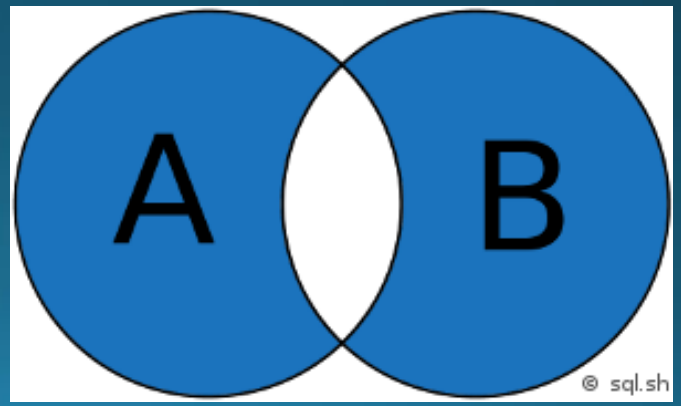
RIGHT JOIN : jointure externe pour retourner les résultats quand la condition est vrai dans au moins une des 2 tables.

```
SELECT *
FROM A
FULL JOIN B ON A.key = B.key
```



(sans intersection)

```
SELECT *
FROM A
FULL JOIN B ON A.key = B.key
WHERE A.key IS NULL
OR B.key IS NULL
```



Pour les autres jointures ce la liste, étant donné que celles-ci sont bien moins utilisées, je vous invite à aller regarder par vous-même les syntaxes sur SQL.SH.

Les Sous Requêtes:

Une sous requête consiste à exécuter une requête à l'intérieur d'une autre requête. Une requête imbriquée est souvent utilisée au sein d'une clause WHERE ou HAVING pour remplacer une ou plusieurs constantes.

```
SELECT *
FROM table1
WHERE nom_attribut1 IN (
SELECT nom_attribut2
FROM table2
WHERE nom_attribut2= " valeur"
);
```