SQL: Bases

Mathieu Gaborit HAUM

Août 2012

Table des matières

Introduction

Mise à part la vanne stupide du titre, ce document est (à peu près) sérieux.

En quoi le titre comporte-t-il une vanne ? Eh bien, nous allons parler des bases du SQL, un langage permettant d'interagir avec les bases de données (vous la comprenez maintenant hein ?).

Ce document n'es **en aucun cas un cours**. Tout au plus une introduction sommaire à un langage pouvant s'avérer complexe et utilisé un peu partout.

Ce qu'on va faire

On abordera les notions principales :
- Connexion au serveur SQL/SGDB
- Création de DB simples et de tables
- Opérations **CRUD** [1]
- Peut être un peu de jointures entre tables

Ce qu'on ne fera pas

- Modélisation de DB au sens propre
- D'admin pure et dure de serveur SQL
- L'explication du fonctionnement interne d'une DB et de son moteur de stockage
- pleins d'autres truc très intéressants [2]

0.1 SQL : kézaco ?

Le **SQL**est un langage apparu dans les années 1970, il est dédiée à l'interaction avec des bases de données **relationnelles**. **SQL**signifie **S**tructured **Q**uery **L**anguage, soit en français dans le texte : Langage de Requête Structurée.

Le langage lui-même peut se décomposer en deux parties : une destinée à gérer les données elles-même et l'autre à administrer la DB [3] hôtesse.

Nous utiliserons dans les tests SQLite [4] qui est un programme pouvant gérer les DB relationnelles sans problème. Les principaux avantages sont que :
- il tourne partout
- il est complet et très léger

1. **CR**eate **U**pdate **D**elete
2. des gateaux au chocolat par exemple
3. DB, database et base de données seront utilisés indifféremment
4. `http://sqlite.org/`

– il ne demande **aucune** configuration
– les DB sont conservées dans un seul et unique fichier

0.2 Installation de SQLite

Vous trouverez tous les liens utiles sur la page de téléchargement [5] mais il existe aussi des paquets pour la plupart des distributions linux principales, ainsi, vous pourrez l'installer via :

```
$ sudo apt-get install sqlite3
```

Sous Ubuntu et Debian et :

```
$ sudo pacman -S sqlite
```

sous ArchLinux.

Si vraiment vous voulez aller plus loin, vous trouverez le code source de SQLite en ligne et pourrez donc le lire/modifier ou le compiler.

Notez que SQLite est à utiliser en ligne de commande (y compris sous Windows et MacOS). Nous utiliserons une version >3.0.0 de SQLite.

Enfin, sachez divers outils existent pour faire joujou avec les DB dont SQlite Manager [6] pour Firefox.

,

0.3 La base de la base : `SELECT`

La partie la plus utilisée des bases de données est quand même l'accès aux données elles-même.

Le mot clé `SELECT` permet de récupérer un ou plusieurs champs d'une table en fonction de critères. Avant de voir comment s'en servir (patience jeune padawan) nous allons détailler un peu le vocabulaire (voir la figure 1).

On note qu'une même DB pourra contenir plusieurs tables elles-même pouvant contenir plusieurs champs. Pour faire un parallèle simple avec le tableur :

– Les DB sont des fichiers
– Les tables sont des tableaux
– Les champs sont les colonnes
– Chaque entrée d'une table est une ligne du tableau

La plupart du temps un de ces champs est unique pour la table et sert donc de clé primaire pour y accèder (`PRIMARY KEY`, parfois abrégé `Primay Key`).

5. `http://sqlite.org/download.html`
6. `https://addons.mozilla.org/fr/firefox/addon/sqlite-manager/`

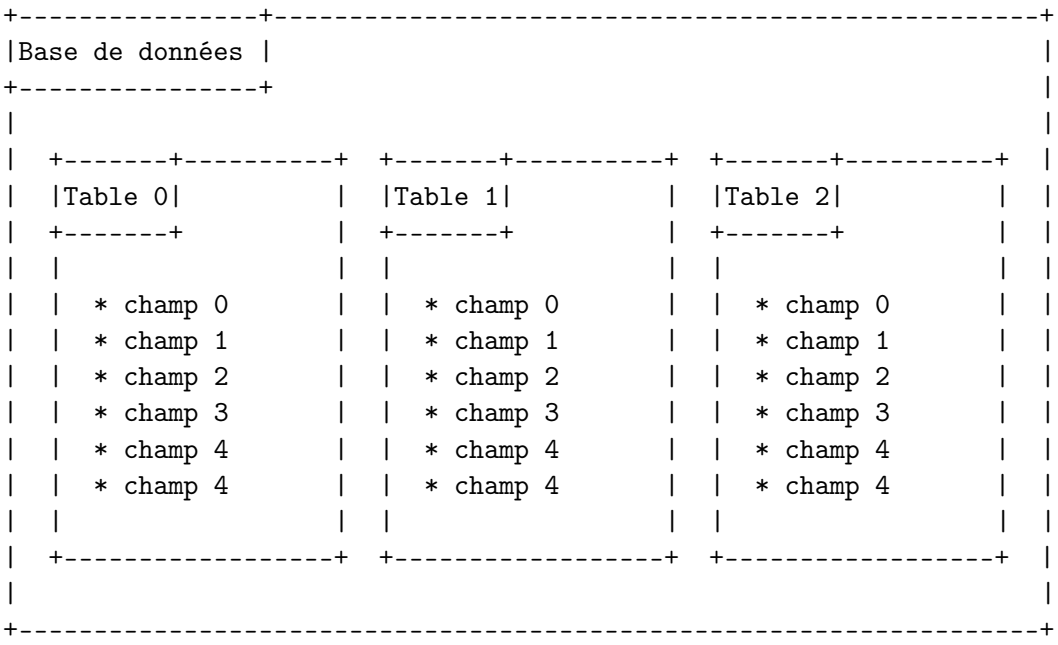

FIGURE 1 – Schema DB/Table/Champ

Cette `Primay Key`est rtès utile pour identifier une ligne de manière unique, le champ servant de `Primay Key`est défini au moment de la création de la table elle même (voir la section **??** en page **??**).

Pour commencer, nous travaillerons sur une DB simple. Copier coller ce qui suit dans un fichier (par exemple nommé `init.sql`) et ouvrez un terminal dans le dossier le contenant :

```
-- CREATE
CREATE TABLE ‘personnes‘ (
    id INTEGER PRIMARY KEY,
    nom VARCHAR,
    prenom VARCHAR,
    surnom VARCHAR,
    titre VARCHAR,
    domicile TEXT,
    enfants INTEGER NOT NULL DEFAULT 0,
    en_couple INTEGER,
UNIQUE (id)
);

-- Just fill it ;)
INSERT INTO ‘personnes‘ (
```

```
    nom,
    prenom,
    surnom,
    titre,
    domicile,
    enfants,
    en_couple
) VALUES (
    'Atréides',
    'Leto',
    '',
    'Duc',
    'Arrakis',
    2,
    1);
INSERT INTO `personnes` (
    nom,
    prenom,
    surnom,
    titre,
    domicile,
    enfants,
    en_couple
) VALUES (
    'Atréides',
    'Paul',
    "Muad'dib",
    'Duc',
    'Arrakis',
    0,
    0);
INSERT INTO `personnes` (
    nom,
    prenom,
    surnom,
    titre,
    domicile,
    enfants,
    en_couple
) VALUES (
    'Atréides',
    'Jessica',
    '',
    'Dame',
```

```
    'Arrakis',
    2,
    1);
INSERT INTO `personnes` (
    nom,
    prenom,
    surnom,
    titre,
    domicile,
    enfants,
    en_couple
) VALUES (
    '',
    'Gandalf',
    'Le Gris',
    '',
    'Terre du Milieu',
    0,
    0);
INSERT INTO `personnes` (
    nom,
    prenom,
    surnom,
    titre,
    domicile,
    enfants,
    en_couple
) VALUES (
    'Dent',
    'Arthur',
    '',
    '',
    'Terre',
    0,
    0);
INSERT INTO `personnes` (
    nom,
    prenom,
    surnom,
    titre,
    domicile,
    enfants,
    en_couple
) VALUES (
```

```
    'Prefect',
    'Ford',
    '',
    '',
    'Betlegeuse 5',
    0,
    0);
INSERT INTO `personnes` (
    nom,
    prenom,
    surnom,
    titre,
    domicile,
    enfants,
    en_couple
) VALUES (
    '',
    'Arya',
    "Tueuse d'Ombre",
    'Dröttning',
    'Ellesmera 5',
    0,
    0);
INSERT INTO `personnes` (
    nom,
    prenom,
    surnom,
    titre,
    domicile,
    enfants,
    en_couple
) VALUES (
    '',
    'Eragon',
    "Tueur d'Ombre",
    'Finiarel',
    'Alagaesia',
    0,
    0);
INSERT INTO `personnes` (
    nom,
    prenom,
    surnom,
    titre,
```

```
    domicile,
    enfants,
    en_couple
) VALUES (
    '',
    'Brom',
    '',
    '',
    'Alagaesia',
    1,
    0);
```

Enseuite, lancez la commande suivante :

```
$ sqlite3 -init init.sql db.sqlite3
```

SQLite devrait vous annoncer qu'il charge les ressources depuis le fichier et vous amener dans un *prompt* une fois le chargement terminé, tapez alors `.quit}`pour en sortir. La base est le fichier `db.sqlite3` qui a été créé.

Vous avez dans ce que vous avez copié/collé une suite de requetes SQL permettant de créer une table nommée `personnes` et d'y insérer plusieurs lignes. Cette table contient des informations sur divers personnage de la littérature fantastique et SF (je remercie d'ailleurs les auteurs [7] d'avoir écrit les bouquins où ces personnages prennent place).

Les informations présentes dans cette table sont :
- le nom du personnage (champ `nom`)
- son prénom (champ `prenom`)
- son surnom (champ `surnom`)
- son titre (champ `titre`)
- son domicile (champ `domicile`)
- le nombre de ses enfants s'il en a 0 sinon (champ `enfants`)
- s'il est en couple (champ `en_couple`, 1 pour oui et 0 pour non)

Les commandes SQL dans la suite sont à lancer dans SQLite, sur la base. Pour cela ouvrez la base avec :

```
$ sqlite3 db.sqlite3
```

SQLite devrait alors vous afficher quelque chose comme ça :

```
SQLite version 3.7.13 2012-06-11 02:05:22
Enter ".help" for instructions
Enter SQL statements terminated with a ";"
sqlite>
```

les commandes seront à taper à la suite de la denière ligne (le *prompt* de SQLite).

7. Frank Herbert, J.R.R. Tolkien, Douglas Adams et Christopher Paolini

0.3.1 La syntaxe de SELECT

Voilà un exemple simple permettant de récuperer tous les champs de toutes les ligne de la table `personnes` :

```
SELECT * FROM personnes;
```

Vous devriez voir apparaitre ceci :

```
1|Atréides|Leto||Duc|Arrakis|2|1
2|Atréides|Paul|Muad'dib|Duc|Arrakis|0|0
3|Atréides|Jessica||Dame|Arrakis|2|1
4||Gandalf|Le Gris||Terre du Milieu|0|0
5|Dent|Arthur|||Terre|0|0
6|Prefect|Ford|||Betlegeuse 5|0|0
7||Arya|Tueuse d'Ombre|Dröttning|Ellesmera 5|0|0
8||Eragon|Tueur d'Ombre|Finiarel|Alagaesia|0|0
9||Brom|||Alagaesia|1|0
```

Expliquons un peu la commande :

```
-- en SQL, les commentaires sont sur une ligne complète,
-- commençant par --

-- Explication de la commande :

-- Sélectionner tous les champs
SELECT *
-- depuis la table personnes
FROM personnes
-- le point virgule est obligatoire en fin de requete
;
```

0.3.2 Limiter les champs à récupérer

On a pas toujours besoin de tous les champs d'une table, ainsi, il y a un moyen de limiter la requète aux champs nous intéressant.

Admettons, que l'on veuille récupérer le prénom et le surnom de tous ceux dans la base, on utilisera alors :

```
SELECT prenom,surnom FROM personnes;
```

Et on obtiendra l'affichage suivant :

```
Leto|
Paul|Muad'dib
Jessica|
Gandalf|Le Gris
Arthur|
Ford|
Arya|Tueuse d'Ombre
Eragon|Tueur d'Ombre
Brom|
```

Il est ainsi possible de donner une liste de champs à récupérer en séparant leur nom par des virgules et en les mettant après `SELECT`. Pour traduire, on pourrait écrire :

```
SELECT {quoi} FROM {quelle table};
```

Notez enfin que l'ordre des champs est important, ainsi, si on lance :

```
SELECT surnom,prenom FROM personnes;
```

on aura :

```
|Leto
Muad'dib|Paul
|Jessica
Le Gris|Gandalf
|Arthur
|Ford
Tueuse d'Ombre|Arya
Tueur d'Ombre|Eragon
|Brom
```

0.3.3 La clause `WHERE`

C'est bien de limiter les champs à récupérer mais parfois, on cherche à récupérer tous les champs des lignes selon un ou plusieurs critères.

`WHERE` permet d'énoncer des conditions et de les combiner entre elles.

Par exemple, si on cherche à récupérer le prénom et le surnom de tous ceux ayant un surnom, on écrira :

```
SELECT prenom,surnom FROM personnes WHERE surnom!='';
```

Ce qui pourrait se traduire par :

> Selectionner prenom et surnom des champs de la table personnes
> si surnom n'est pas une chaine vide

Cette commande produira l'affichage suivant :

```
Paul|Muad'dib
Gandalf|Le Gris
Arya|Tueuse d'Ombre
Eragon|Tueur d'Ombre
```

On peut utiliser plusieurs symboles pour les tests :

= égalité

!= inégalité

> supériorité

< infériorité

>= supériorité ou égalité

<= infériorité ou égalité

Exercice Essayez d'écrire une requète pour récupérer le titre puis le prénom et enfin le nom de toute la famille Atréides.

Réponse :

```
SELECT titre,prenom,nom FROM personnes WHERE nom='Atréides';
```

Exercice Essayez d'écrire une requète pour récupérer le prénom de tous ceux ayant au moins un enfant.

Réponse :

```
SELECT prenom FROM personnes WHERE enfant>=1;
```

Exercice Essayez d'écrire une requète pour récupérer le prénom de tous ceux ayant plus d'un enfant.

Réponse :

```
SELECT prenom FROM personnes WHERE enfant>1;
```

0.3.4 SQL a les `BOOL`s

Ahem. Bon, admettons, il y aura au moins eu deux blagues vaseuses...

Maintenant qu'on sait faire de jolies requètes avec des morceaux de `WHERE` dedans, essayons de combiner un peu.

Lancez ces deux requètes et observez les résultats.

```
SELECT prenom FROM personnes WHERE titre!='' AND surnom !='';
SELECT prenom FROM personnes WHERE titre!='' OR surnom !='';
```

Que pouvez vous en déduire ?

Le mot clé `OR` réalise un **ou logique** entre les conditions de part et d'autre, pour `AND`, il s'agit d'un **et logique**.

Notez qu'on peut augmenter les combinaisons :

```
SELECT prenom,enfants
FROM personnes
WHERE (titre!='' OR surnom !='') AND enfants!=0;
```

nous renverra ainsi :

```
Leto|2
Jessica|2
```

Pour traduire, voilà ce que nous avons demandé :

> Sélectionner prenom et nombre d'enfants de tout ceux ayant des enfants et soit un surnom, soit un titre, ou les deux.

Remarquez que Brom a un enfant mais que sans titre ni surnom, il ne peut apparaitre dans cette liste. Vous pouvez le vérifier avec la requète suivante :

```
SELECT prenom,titre,surnom,enfants FROM personnes WHERE prenom="Brom";
```

0.3.5 `ORDER BY` : trier les résultats

Il peut être intéressant de trier les résultats d'une requète.

Pour cela il y a `ORDER BY` ainsi que :

`ASC` Tri ascendant (défaut)

`DESC` Tri descendant

Ainsi pour avoir la liste des prénoms de toutes les personnes dans l'ordre anti-alphabétique, on écrira :

```
SELECT prenom FROM personnes ORDER BY prenom DESC;

-- pour l'order alphabétique :
SELECT prenom FROM personnes ORDER BY prenom ASC;
-- ou :
SELECT prenom FROM personnes ORDER BY prenom;
```

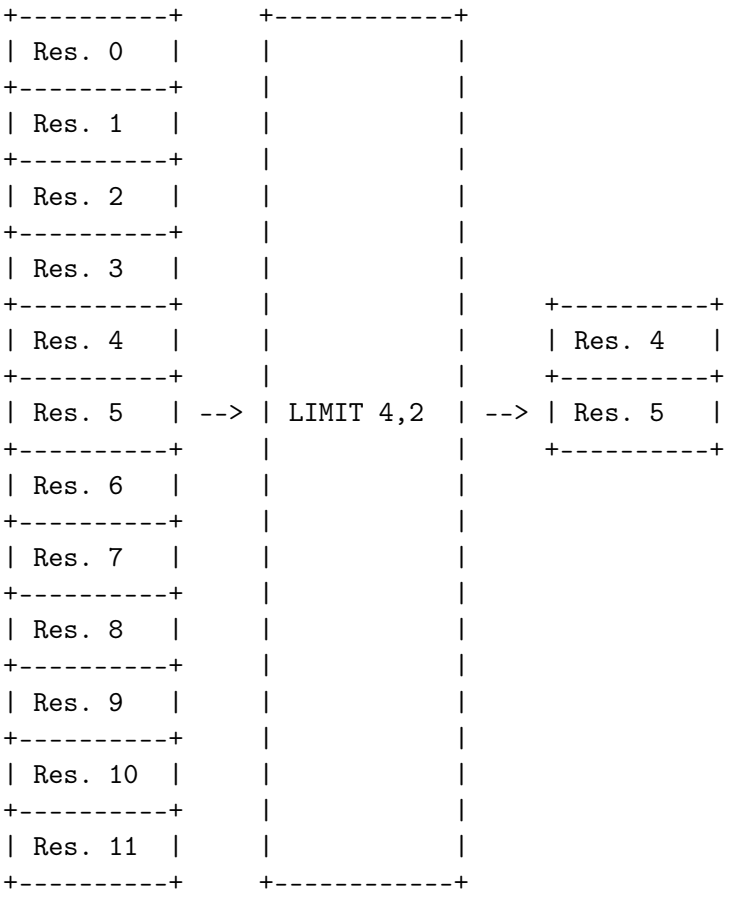

FIGURE 2 – Principe du LIMIT

0.3.6 Limiter le nombre de résultats avec `LIMIT`

Nous savons maintenabt choisir les champs à récupérer, trier et placer des critères.

Il nous reste à apprendre à limiter le nombre de résultats : c'est là qu'intervient le mot clé `LIMIT`.

La syntaxe est simple :

```
<requete> LIMIT [OFFSET,] NOMBRE;
```

`LIMIT` prend a deux arguments possibles : le nombre de résultats à envoyer et un offset, le nombre de résultats à bloquer avant de commencer le compte, voir la figure 2.

Si l'offset est nul (que le décompte commence au premier résultat, la partie avant la virgule peut être omise. Pour les 10 premiers résultats d'une requète, on écrira :

```
<requete> LIMIT 10;
```

Exercice Essayez d'écrire une requète pour récupérer les ids des trois premières personnes dans l'ordre alphabétique des prénoms.

Réponse :

```
SELECT id FROM personnes ORDER BY prenom LIMIT 3;
```

0.4 Ajouter des lignes avec `INSERT`

Dans la suite des commandes du `CRUD`, `INSERT` est incontournable.

Pour l'instant, nous avons travaillé à récupérer des données dans une base existante. C'est bien, mais à un moment, il nous a fallu remplir cette DB. Pour comprendre un peu mieux le `INSERT`, regardons un des blocs SQL que je vous ai donné au début pour la création et le remplissage initial de la base :

```
INSERT INTO ‘personnes‘ (
    nom,
    prenom,
    surnom,
    titre,
    domicile,
    enfants,
    en_couple
) VALUES (
    ’Atréides’,
    ’Leto’,
    ’’,
    ’Duc’,
    ’Arrakis’,
    2,
    1);
```

Ce bloc montre bien la structure du `INSERT` :

```
INSERT INTO nom_de_la_table (
    liste,
    de,
    champs
) VALUES (
    ’valeurs’,
    ’qui’,
    ’correspondent’);
```

Bien sûr, la requête peut être écrite sur une seule ligne `;)`.

Plusieurs choses à noter :

- le nombre de champs et de valeurs doivent être les même sinon, une erreur se produira
- les backquotes présentes autour du nom de la table dans le premier code permet de forcer l'interprétation comme une chaine faisant partie de la commande (utile dans certains cas).
- le champ `id` présent dans la définition de la table n'est pas renseigné ici mais peut importe : il s'auto incrémente
- dans le cas d'ajout automatisés (via un script, à partir d'une entrée de l'utilisateur par exemple) il faut veiller à ce qu'aucune quote non échappée ne subsiste dans les chaines des valeurs. Il s'agit d'une faille de sécurité courante permettant de réaliser des injections et d'altérer la base de donnée. La plupart des langages proposent des fonctions pour réaliser ce travail.

0.4.1 Rapide retour sur les problèmes de sécurité

Dans le cas de scripts PHP par exemple, le fait de laisser des chaines non échappés peut suffir à créer une bréche énorme dans l'application complète. Vous vous exposez alors à diverses fuites de données et risques de modification.

Pour cous entrainer, essayez d'ajouter quelques personnes à notre base. Essayez ensuite de selectionner les nouveaux champs via des requetes.j

0.5 `DELETE` : supprimer des enregistrements

Nous connaissons maintenant la base de l'ajout et récupération de données, voyons comment les supprimer.

Le mot clé est `DELETE` et il travaille forcément sur la ligne complète.

On peut l'associer aux diverses clauses que l'on a vu plus tot pour cibler son action.

Ainsi :

```
DELETE FROM personnes WHERE enfants=0;
```

Supprimera tous les enregistrements de personnes n'ayant pas d'enfant.

Pour vous entrainer, essayer de supprimer des éléments selon divers critères. Recréez la base comme indiqué au début du document une fois que vous aurez bien pris `DELETE` en main.

0.6 Mise à jour : `UPDATE` time !

Même si le fait de récupèrer, supprimer et réinsérer un enregistrement suffit théoriquement à le "mettre à jour", c'est une technique particulièrement sale, surtout quand on se souvient que `UPDATE` est facile à maitriser.

La syntaxe générale est la suivante :

```
UPDATE nom_de_la_table SET nom_du_champ=valeur <clauses>;
```

Où `<clauses>` est à replacer par des `WHERE` et autres trucs.

On pourrait par exemple envisager de rajouter le titre "parent" à toutes les personnes ayant des enfants mais pas de titre. Cela ne devrait affecter que Brom :

```
sqlite> SELECT id,prenom,enfants,titre FROM personnes WHERE prenom='Brom';
9|Brom|1|
sqlite> UPDATE personnes SET titre='parent' WHERE enfants!=0 AND titre='';
sqlite> SELECT id,prenom,enfants,titre FROM personnes WHERE prenom='Brom';
9|Brom|1|parent
```

`UPDATE` est donc un excellent moyen de modifier les enregistrements et pour peu que vous maitrisiez bien les clauses de sélection, l'utilisation ne devrait poser aucun problème.

0.7 Création : `CREATE`

La création de tables et de DB est soumise à plusieurs contraintes. La plus importante est que les types primitifs peuvent varier selon les SGDB et poser ainsi des problèmes de compatibilité.

0.7.1 Création de `DATABASE`

Nous ne rentrerons pas dans les aspects complets de la chose, il faudrait un livre.

Sachez simplement que la plus simple expression pour créer une base de données est :

```
CREATE DATABASE nom_de_la_db;
```

Pour SQLite, c'est un jeu d'enfant : un même fichier ne peut contenir qu'une seule et unique base.

0.7.2 Création de `TABLE`

La création de table est un peu plus intéressante, et même si elle est aussi sujette à variation, elle reste plus utile que la création de DB seules.

Là encore on utilise :

```
CREATE TABLE nom_de_la_table
-- ....
```

mais cette fois, on fait suivre l'assertion d'un groupe parenthèsé décrivant les champs inclus (nom, type et attributs).

Par exemple, pour la DB utilisée tout au long du document :

```
CREATE TABLE ‘personnes‘ (
    id INTEGER PRIMARY KEY,
    nom VARCHAR,
    prenom VARCHAR,
    surnom VARCHAR,
    titre VARCHAR,
    domicile TEXT,
    enfants INTEGER NOT NULL DEFAULT 0,
    en_couple INTEGER,
UNIQUE (id)
);
```

`VARCHAR` désigne un champ destiné à contenir des caractères, `INTEGER` des entiers. Pour `TEXT` il s'agit de texte plus long, souvent avec des retour chariot. Sachez qu'il existe aussi `DATE`, `DATETIME`, pour les dates (avec ou sans heures).

Enfin, plusieurs "tailles" existent dans certains SGDB, MySQL propose ainsi : `TINYINT`, `MEDIUMINT`, `BIGINT`, etc...

Dans une application utilisant MySQL, j'ai utilisé le schéma suivant :

```
create database if not exists dataporn;
create table if not exists ‘dataporn‘.‘demande‘ (
    -- general
    id bigint not null auto_increment,
    valide tinyint(1) default 0,

    -- perso
    nom varchar(255) not null,
    email varchar(255) not null,
    adresse text not null,
    note text,
```

```
    -- stickers
    qte mediumint not null,
    prix float not null,

    -- hackerspaces et repartition
    adhaum tinyint(1) not null default 0,
    adipefix tinyint(1) not null default 0,
    montant_haum float not null,
    montant_ipefix float not null,

    -- primary key
    primary key(id)
);
```

Ainsi vous pouvez voir plusieurs choses :

- Les commentaires sont utiles (on le répétera jamais assez)
- la clause `IF NOT EXIST` permet de poser une condition à la création d'une DB/table (et d'éviter les erreurs)
- `TINYINT(1)` est utilisé en lieu et place du type `BOOL` proposé par MySQL
- la déclaration du champ auto-incrémenté (`id`) est différent entre les deux SGDB.
- les majuscules dans les requètes (pour les mots-clés) ne sont pas du tout obligatoire (mais c'est plus clair quand on débute).
- pour désigner la table `demande` de la DB `dataporn` j'ai utilisé `` `dataporn`.`demande` ``. Ici, les backquotes prennent tout leur sens.

Si vous vous demandez pour qu'elle application est ce schéma : il s'agit d'un moyen potentiel de financer les hackerspaces du Mans et d'Angers [8].

8. `https://github.com/manudwarf/dataporn`