Chapitre 3

Environnement MySQL/PHP

Sommaire

3.1 Apports de MySQL et PHP 41
3.1.1 MySQL 42
3.1.2 PHP 44
3.2 Une première base MySQL 46
3.2.1 Création d'une table 46
3.2.2 L'utilitaire *mysql* 47
3.2.3 L'interface phpMyAdmin 53
3.3 Accès à MySQL avec PHP 55
3.3.1 L'interface MySQL/PHP 56
3.3.2 Formulaires d'interrogation 61
3.3.3 Formulaires de mises à jour 63

Nous décrivons maintenant l'utilisation de MySQL et de PHP. Comme dans le chapitre précédent, nous nous limiterons pour l'instant à une application très simple, consistant à gérer sur le serveur une liste de films et à proposer aux utilisateurs de consulter une partie de ces films, sélectionnés selon des critères de recherche entrés dans un formulaire, et présentés dans un document HTML.

Cette fonctionnalité est identique à celle que nous avons développée (avec quelques efforts) dans le chapitre précédent, sous la forme d'un programme C basé sur l'interface CGI. En fait la programmation de sites web avec MySQL/PHP est une variante des techniques CGI. Dans les deux cas toute la complexité est du côté du serveur, le client ne recevant que des documents HTML qu'il peut se contenter d'afficher. Dans les deux cas on combine des techniques de programmation (C ou PHP) et l'accès à des sources de données extérieures au programme (un fichier texte dans notre exemple du chapitre précédent ou une base MySQL dans tout ce qui suit).

3.1 Apports de MySQL et PHP

Toute l'avantage de la combinaison MySQL/PHP par rapport à une solution CGI classique tient dans la puissance est la facilité d'utilisation de ces outils. La comparaison entre les deux techniques montre que l'environnement MySQL/PHP nous permet de faire beaucoup mieux et beaucoup plus ! Il est en fait assez facile de se convaincre que l'approche adoptée par le programme CGI du chapitre 2 souffre de nombreuses faiblesses et que beaucoup d'efforts seraient nécessaires pour obtenir une application tenant la route.

Les problèmes découlent en premier lieu de l'utilisation directe des fichiers texte :

Rigidité du format. Si vous regardez attentivement le fichier *films.txt* que nous avons utilisé dans le chapitre 2, vous constaterez que son format est très contraint. Par exemple le titre du film consiste en un seul mot et toutes les caractéristiques des films sont connues pour tous les films. Si ces contraintes ne sont pas respectées, le programme ne marche plus !

Lourdeur d'accès aux données. En pratique, pour chaque accès, même le plus simple, il faudrait écrire un programme qui ouvre le fichier, parcourt ses lignes, effectue des tests spécifiques, etc. Si on s'autorise (ce qui est indispensable) plusieurs mots pour un titre de film, ou si on accepte d'ignorer, par exemple, le prénom du metteur en scène, il faut définir un format complexe, difficile à analyser.

Manque de sécurité. Si tout programmeur peut accéder directement aux fichiers, il est impossible de garantir la sécurité et l'intégrité des données. Tout devient possible, par exemple un programme qui ouvre le fichier des films pour y écrire les cours de la bourse...

Pas de contrôle de concurrence. Imaginons que l'application permette d'insérer de nouveaux films (ce que nous allons d'ailleurs faire dans ce qui suit). Que va-t-il se passer si plusieurs utilisateurs accèdent simultanément au même fichier ?

Tous ces problèmes sont reconnus comme étant très difficiles à résoudre (essayez si vous n'êtes pas convaincus...). Il existe donc des logiciels spécialisés qui prennent en charge tous les problèmes d'accès aux données stockées dans un ou plusieurs fichiers, plus généralement appelées *bases de données*. Ces logiciels sont les *Systèmes de gestion de Bases de Données*, (SGBD), et MySQL en est un représentant.

3.1.1 MySQL

Le recours à MySQL permet de masquer les détails complexes et fastidieux liés à l'utilisation de fichiers. MySQL gère pour vous les fichiers constituant une base de données, prend en charge les fonctionnalités de protection et de sécurité et fournit un ensemble d'interfaces de programmation (dont une avec PHP) facilitant l'accès aux données.

La complexité de logiciels comme MySQL est due à la diversité des techniques mises en œuvre, à la multiplicité des composants intervenant dans leur architecture, et également aux différents types d'utilisateurs (administrateurs, programmeurs, non informaticiens, ...) qui sont confrontés, à différents niveaux, au système. Au cours de ce livre nous aborderons ces différents aspects, tous ne vous étant d'ailleurs pas utiles, en particulier si votre objectif n'est pas d'administrer une base MySQL. Pour l'instant, nous nous contenterons de décrire l'essentiel, à savoir son architecture et ses composants.

MySQL consiste en un ensemble de programmes qui sont chargés de gérer une ou plusieurs bases de données, et qui fonctionnent selon une architecture client/serveur (voir figure 3.1).

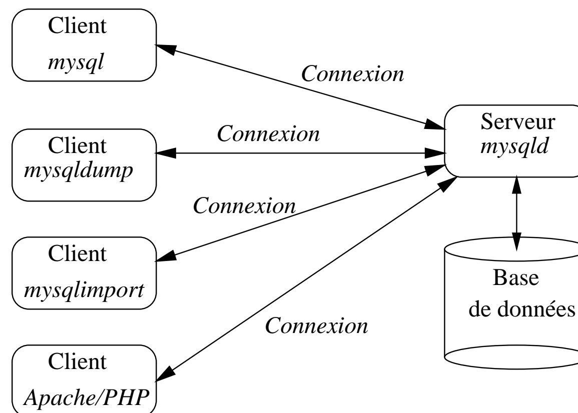

FIG. 3.1 – *Serveur et clients de MySQL.*

Le serveur *mysqld*. Le processus *mysqld* est le serveur de MySQL. Lui seul peut accéder aux fichiers stockant les données pour lire et écrire des informations.

Utilitaires. MySQL fournit tout un ensemble de programmes, que nous appellerons *utilitaires* par la suite, qui sont chargés de dialoguer avec *mysqld*, par l'intermédiaire d'une *connexion*, pour accomplir un type de tâche particulier. Par exemple *mysqldump* permet d'effectuer des sauvegardes, *mysqlimport* peut importer des fichiers ASCII dans une base, etc. Nous étudierons les utilitaires de MySQL dans

le chapitre 8. Le client le plus utile est simplement nommé *mysql*, et permet d'envoyer directement des commandes au serveur.

La base de données est un ensemble de fichiers stockant les informations selon un format propre à MySQL et qui peut – doit – rester inconnu à l'utilisateur. Le serveur est le seul habilité à lire/écrire dans ces fichiers, en fonction de demandes effectuées par des clients MySQL. Noter qu'il peut y avoir plusieurs clients accédant simultanément à une même base. Le serveur se charge de coordonner ces accès.

Les clients de MySQL communiquent avec le serveur pour effectuer des recherches ou des mises à jour dans la base. Cette communication n'est pas limitée à des processus situés sur la même machine : il est possible de s'adresser au serveur MySQL par un réseau comme l'Internet, même si nous n'utiliserons pas cette possibilité.

Il est possible de créer soi-même son propre client MySQL en utilisant des outils de programmation qui se présentent sous la forme d'un ensemble de fonctions, habituellement désigné par l'acronyme API pour *Application Programming Interface*. MySQL fournit une API en langage C, à partir de laquelle plusieurs autres ont été créées, dont une API en PHP. Comme tous les autres clients de MySQL, un script PHP en association avec Apache doit établir une connexion avec le serveur pour pouvoir dialoguer avec lui et rechercher ou mettre à jour des données (figure 3.1).

Bases de données relationnelles

MySQL est un SGBD *relationnel*, comme beaucoup d'autres dont ORACLE, SYBASE, SQL/Server, etc. Le point commun de tous ces systèmes est de proposer une représentation extrêmement simple de l'information sous forme de *table*. Voici une table relationnelle *Film*, montrant les quelques films que nous avons manipulés jusqu'à présent.

titre	année	nomMES	prénomMES	annéeNaiss
Alien	1979	Scott	Ridley	1943
Vertigo	1958	Hitchcock	Alfred	1899
Psychose	1960	Hitchcock	Alfred	1899
Kagemusha	1980	Kurosawa	Akira	1910
Volte-face	1997	Woo	John	1946
Pulp Fiction	1995	Tarantino	Quentin	
Titanic	1997	Cameron	James	1954
Sacrifice	1986	Tarkovski	Andrei	1932

Il y a quelques différences essentielles entre cette représentation et le stockage dans un fichier. D'une part les informations sont conformes à une description précise. Ici la table s'appelle *Film*, et elle comprend un ensemble d'attributs comme *titre*, *année*, etc. Une base de données est constituée d'une ou plusieurs tables, dont les descriptions sont connues et gérées par le serveur. Nous verrons qu'il est possible, *via* un langage simple, de spécifier le format d'une table, ainsi que le type des attributs et les contraintes qui s'appliquent aux données comme, par exemple : *il ne doit pas y avoir deux films avec le même titre*. Tout ce qui concerne la description des données, et pas les données elles-mêmes, constitue le *schéma* de la base de données.

Si on regarde maintenant la représentation des données, on constate qu'elle est beaucoup plus souple et puissante que le simple fichier que nous avions utilisé. Il est possible d'introduire des caractères blancs dans les titres des films (voir 'Pulp Fiction') ou de laisser une cellule de la table vide si on ignore son contenu (comme par exemple l'année de naissance de Tarantino).

Les SGBD relationnels offrent non seulement une représentation simple et puissante, mais également un langage, SQL, pour interroger ou mettre à jour les données. SQL est incomparablement plus facile à utiliser qu'un langage de programmation classique comme le C. Voici par exemple comment on demande la liste des titres de film parus après 1980.

```
SELECT     titre
FROM       Film
WHERE      annee >1980
```

Cette approche est très simple puisqu'elle se contente d'indiquer ce que l'on veut obtenir, à charge pour le SGBD de déterminer *comment* on peut l'obtenir. SQL est un langage *déclaratif* qui permet d'interroger une base sans se soucier de la représentation interne des données, de leur localisation, des chemins d'accès ou des algorithmes nécessaires. À ce titre il s'adresse à une large communauté d'utilisateurs potentiels (pas seulement des informaticiens) et constitue un des atouts les plus spectaculaires (et le plus connu) des SGBD relationnels. On peut l'utiliser de manière interactive, mais également en association avec des interfaces graphiques, des outils de *reporting* ou, très généralement, des langages de programmation.

Ce dernier aspect est important en pratique car SQL ne permet pas de faire de la programmation au sens courant du terme et doit donc être associé avec un langage comme le C, ou PHP.

3.1.2 PHP

Le langage PHP a été créé par Rasmus Lerdorf vers la fin de l'année 1994, pour ses besoins personnels. Comme dans beaucoup d'autres cas, la mise à disposition du langage sur l'Internet est à l'origine de son développement par beaucoup d'autres utilisateurs qui y ont vu un outil propre à satisfaire leurs besoins. Après diverses évolutions, PHP en est (depuis le mois de mai 2000) à sa version 4, celle que nous utilisons. Au moment où ce livre est écrit, l'ordre de grandeur du nombre de sites utilisant PHP (souvent en association avec MySQL) est estimé à 2-3 millions et semble en progression constante.

Qu'est-ce que PHP

PHP est un langage de programmation, très proche du langage C – dont il reprend l'essentiel de la syntaxe – et destiné à être intégré dans des pages HTML. Contrairement à d'autres langages, PHP est exclusivement dédié à la production de pages HTML générées dynamiquement. Voici un premier exemple.

Exemple 10 *ExPHP1.php* : *Premier exemple PHP*

```
<HTML> <HEAD>
<TITLE>HTML avec PHP</TITLE>
<LINK REL=stylesheet HREF="films.css" TYPE="text/css">
</HEAD>
<BODY>

<H1>HTML + PHP</H1>

Nous sommes le <?php   echo Date ("j/m/Y");  ?>

<P>

<?php
  echo "Je suis $HTTP_USER_AGENT et je dialogue avec $SERVER_NAME.";
?>

</BODY></HTML>
```

Il s'agit d'un document contenant du code HTML classique, au sein duquel on a introduit des commandes encadrées par les balises `<?php` et `?>`. Tout ce qui se trouve entre ces commandes est envoyé à un interpréteur du langage PHP intégré à Apache. Cet interpréteur lit les instructions et les exécute.

Ici on a deux occurrences de code PHP (que l'on appellera « scripts » à partir de maintenant). La première fait partie de la ligne suivante :

```
Nous sommes le <?php echo Date ("j/m/Y");?>
```

Le début de la ligne est du texte traité par le serveur Apache comme du HTML. Ensuite on trouve une instruction `echo Date ("j/m/Y");`. La fonction *echo* est l'équivalent du *printf* utilisé en langage C. Elle écrit sur la sortie standard, laquelle est directement transmise au navigateur par le serveur web. *Date* est une fonction PHP qui récupère la date courante et la met en forme selon un format donné (ici, la chaîne `j/m/Y` qui correspond à jour, mois et année sur quatre chiffres).

La syntaxe de PHP est relativement simple, et la plus grande partie de la richesse du langage réside dans ses innombrables fonctions. Il existe des fonctions pour créer des images, pour générer du PDF, pour lire ou écrire dans des fichiers, et – ce qui nous intéresse particulièrement – pour accéder à des bases de données.

Remarque : Le langage PHP est introduit progressivement à l'aide d'exemples. Si vous souhaitez avoir dès maintenant un aperçu complet du langage, reportez-vous à l'annexe C.

Le script *ExPHP1.php* illustre un autre aspect du langage. Non seulement il s'intègre directement avec le langage HTML, mais toutes les variables d'environnement décrivant le contexte des communications entre le navigateur et le serveur web sont directement accessibles sous forme de variables PHP.

```
echo "Je suis $HTTP_USER_AGENT et je dialogue avec $SERVER_NAME.";
```

Tous les noms de variable de PHP débutent par un '$'. L'exemple montre que l'on dispose automatiquement de toutes les variables que nous avons citées au moment de la présentation de l'interface CGI (voir table 2.1, page 35), entre autres, de `HTTP_USER_AGENT` qui donne le nom du navigateur, et de `SERVER_NAME` qui donne le nom du serveur. Vous pouvez remarquer que ces variables peuvent être directement utilisées dans une chaîne de caractères délimitée par (").

PHP est du côté serveur

Il est essentiel d'être bien conscient qu'un script PHP est exécuté par un interpréteur qui se trouve *du côté serveur*. En cela PHP est complètement différent d'un langage comme JavaScript qui s'exécute sur le navigateur. En général l'interpréteur PHP est intégré à Apache sous forme de module, et le mode d'exécution est alors très simple. Quand un fichier avec une extension `.php` (ou `.php3` pour les anciennes versions) est demandé au serveur web, ce dernier le charge en mémoire et y cherche tous les scripts PHP qu'il transmet à l'interpréteur. L'interpréteur exécute le script, ce qui a pour effet de produire du code HTML qui vient remplacer le script PHP dans le document finalement fourni au navigateur. Ce dernier reçoit donc du HTML « pur » et ne voit jamais la moindre instruction PHP.

À titre d'exemple, voici le code HTML produit par le fichier PHP précédent, tel que vous pouvez vous-mêmes le vérifier sur notre site. Le résultat correspond à une exécution sur la machine serveur *cartier.cnam.fr*, avec un navigateur Netscape (dont le nom est Mozilla). Les parties HTML sont inchangées, les scripts PHP ont été remplacés par le résultat de leur exécution.

```
<HTML> <HEAD>
<TITLE>HTML avec PHP</TITLE>
<LINK REL=stylesheet HREF="films.css" TYPE="text/css">
</HEAD>
<BODY>

<H1>HTML + PHP</H1>

Nous sommes le 20/09/2000
<P>

Je suis Mozilla/4.72 [en] (X11; I; Linux 2.2.14 i686)
et je dialogue avec cartier.cnam.fr.
</BODY></HTML>
```

Le principe est donc très proche du CGI, avec des améliorations notables :

1. on peut mixer HTML et PHP de manière très souple ;
2. les variables CGI ou autres sont fournies directement et sans effort (comparer avec le programme C du chapitre précédent) ;
3. enfin les scripts sont exécutés directement au sein d'Apache, ce qui évite d'avoir à lancer systématiquement un programme CGI.

Accès à MySQL

Un des grands atouts de PHP est sa très riche collection d'interfaces (API) avec tout un ensemble de SGBD. En particulier il est possible à partir d'un script PHP de se connecter à un serveur *mysqld* pour récupérer des données que l'on va ensuite afficher dans des documents HTML. D'une certaine manière, PHP permet de faire d'Apache un client MySQL, ce qui aboutit à l'architecture de la figure 3.2.

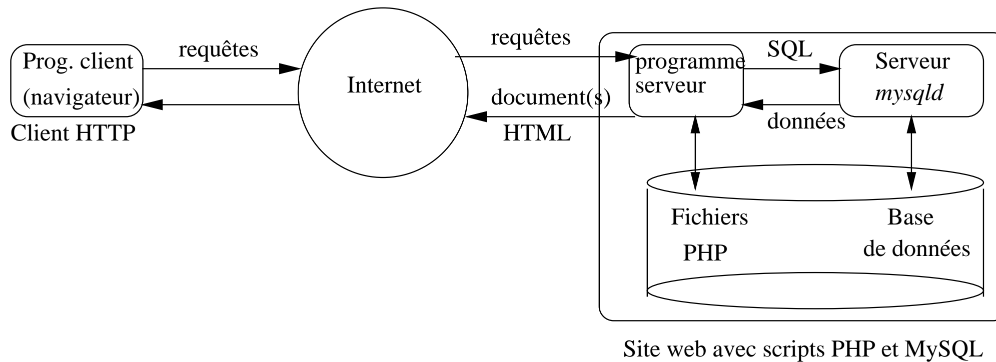

FIG. 3.2 – *Architecture d'un site web avec MySQL/PHP*

Il s'agit d'une architecture à trois composantes, chacune réalisant une des trois tâches fondamentales d'une application.

1. le *navigateur* constitue *l'interface graphique* dont le rôle est de permettre à l'utilisateur de visualiser et d'interagir avec l'information ;
2. MySQL est *le serveur de données* ;
3. enfin l'ensemble des fichiers PHP contenant le code d'extraction, traitement et mise en forme des données est le *serveur d'application*, associé à Apache qui se charge de transférer les documents produits sur l'Internet.

Rien n'empêche d'aller un tout petit peu plus loin et d'imaginer une architecture où les trois composantes sont franchement séparées et dialoguent par l'intermédiaire du réseau Internet. Ici nous supposerons que le serveur *mysqld* et Apache sont sur la même machine, mais le passage à une solution réellement à « trois pôles » ne présente pas, ou peu, de différence du point de vue technique.

3.2 Une première base MySQL

Nous allons maintenant mettre ces principes en application en créant une première base MySQL contenant notre liste de films, et en accédant à cette base avec PHP. Pour l'instant nous présentons les différentes commandes d'une manière simple et intuitive avant d'y revenir plus en détail dans les prochains chapitres.

3.2.1 Création d'une table

La première base, très simple, est constituée d'une seule table *FilmSimple*, avec les quelques attributs déjà rencontrés précédemment. Pour créer des tables, on utilise une partie de SQL dite « Langage de Définition de Données » (DDL) dont la commande principale est le `CREATE TABLE`.

```
CREATE TABLE FilmSimple
  (titre VARCHAR (30),
  annee INTEGER,
  nomMES VARCHAR (30),
  prenomMES VARCHAR (30),
  anneeNaiss INTEGER
  );
```

La syntaxe du `CREATE TABLE` se comprend aisément. On indique le nom de la table, qui sera utilisé par la suite pour accéder à son contenu, puis la liste des attributs avec leur type. Pour l'instant nous nous en tenons à quelques types de base : `INTEGER`, que l'on peut abréger en `INT`, est un entier, et `VARCHAR` est une chaîne de caractères de longueur variable, pour laquelle on spécifie la longueur maximale.

Remarque : On peut utiliser indifféremment les majuscules et les minuscules pour les mot-clé de SQL. De même, les sauts de ligne, les tabulations et les espaces successifs dans un ordre SQL équivalent à un seul espace pour l'interpréteur et peuvent donc être utilisés librement pour clarifier la commande.

Pour exécuter une commande SQL, il existe plusieurs possibilités, la plus générale étant d'utiliser le client *mysql* dont le rôle est principalement celui d'un interpréteur de commande. C'est la solution que nous adopterons dans la plupart des cas.

Quand vous avez recours aux services d'un fournisseur d'accès à distance, ce dernier propose généralement d'entrer des commandes par l'intermédiaire d'une interface web d'administration de bases MySQL, *phpMyAdmin*. Cet outil est – à juste titre – très populaire, et fournit un environnement de travail *graphique*, donc plus convivial que l'interpréteur de commandes *mysql*.

Nous envisageons successivement les deux situations, *mysql* et *phpMyAdmin*, dans ce qui suit.

3.2.2 L'utilitaire *mysql*

Si vous avez suivi le processus d'installation et de configuration décrit dans l'annexe A, vous devriez déjà disposer d'un accès (nom et mot de passe) au client *mysql* et d'une base *Films*. Voici un rappel de l'essentiel du processus de connexion.

Connexion

Voici tout d'abord comment créer une base de données et un nouvel utilisateur avec l'utilitaire *mysql*.

```
% mysql -u root -p
Enter password:
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 6 to server version: 3.23.9

Type 'help' for help.

mysql> CREATE DATABASE Films;
mysql>
mysql> GRANT ALL PRIVILEGES ON Films.* TO rigaux@localhost
            IDENTIFIED BY 'mdprigaux';
mysql> exit
```

Le prompt `mysql>` est celui de l'interpréteur de commandes de MySQL : ne pas entrer de commandes Unix à ce niveau, et réciproquement !

La commande `CREATE DATABASE` crée une base de données *Films*, autrement dit un espace dans lequel on va placer une ou plusieurs tables contenant les données de l'application. La commande `GRANT` définit un utilisateur `rigaux` qui aura tous les droits (`ALL PRIVILEGES`) pour accéder à cette base et manipuler ses données. On peut alors se connecter à la base *Films* sous le compte `rigaux` avec :

```
% mysql -u rigaux -p Films
```

L'option `-p` indique que l'on veut entrer un mot de passe. *mysql* affiche alors un prompt `password:`

Il est possible de donner le mot de passe directement après `-p` dans la ligne de commande mais ce n'est pas une très bonne habitude à prendre que d'afficher en clair des mots de passe.

La meilleure méthode consiste à stocker dans un fichier de configuration le compte d'accès à MySQL et le mot de passe. Pour éviter à l'utilisateur `rigaux` d'entrer systématiquement ces informations, on peut

ainsi créer un fichier *.my.cnf* dans le répertoire `$HOME` (sous Unix) ou *C:* (sous Windows), et y placer les informations suivantes :

```
[client]
user= rigaux
password = mdprigaux
```

Tous les programmes clients de MySQL lisent ce fichier et utiliseront ce compte d'accès pour se connecter au programme serveur *mysqld*. La connexion à la base *Films* devient alors simplement :

```
% mysql Films
```

Bien entendu, il faut s'assurer que le fichier *.my.cnf* n'est pas lisible par les autres utilisateurs, ce qui est facile sous Unix, et impossible sous Windows. Nous renvoyons à l'annexe A pour plus de détails sur l'utilisation des fichiers de configuration.

Création de la table

On suppose maintenant que l'utilisateur dispose d'un fichier de configuration. Voici la séquence complète de commandes pour créer la table *FilmSimple*.

```
% mysql

Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 10 to server version: 3.23.9

Type 'help' for help.

mysql> USE Films;
Database changed
mysql> CREATE TABLE FilmSimple
    ->   (titre      VARCHAR (30),
    ->    annee      INTEGER,
    ->    nomMES     VARCHAR (30),
    ->    prenomMES  VARCHAR (30),
    ->    anneeNaiss INTEGER
    ->   );
Query OK, 0 rows affected (0.01 sec)
mysql> exit
```

La commande `USE Films` indique que l'on s'apprête à travailler sur la base *Films* (rappelons qu'un serveur peut gérer plusieurs bases de données). On peut de manière équivalente, donner le nom de la base sur la ligne de commande de *mysql*. La commande `CREATE TABLE`, entrée ligne à ligne, provoque l'affichage de `->` tant qu'un point-virgule indiquant la fin de la commande n'est pas entré au clavier.

Remarque : Peut-on utiliser des accents dans les noms de table et d'attributs ? La réponse est *oui*, du moins si MySQL a été installé en précisant que le jeu de caractères à utiliser est du type *Latin*. Il y a quand même un petit problème à prendre en compte : ces noms sont utilisés pour interroger la base de données, et tous les claviers ne disposent pas des caractères accentués. Nous avons pris le parti de ne pas utiliser d'accent pour tout le code informatique. En revanche les informations stockées dans la base pourront elles contenir des accents. À vous de juger du choix qui convient à votre situation.

La table *FilmSimple* est maintenant créée. Vous pouvez consulter son schéma avec la commande `DESC` et obtenir l'affichage ci-dessous. Seules les informations `Field` et `Type` nous intéressent pour l'instant (pour des raisons obscures, MySQL affiche `int(11)` au lieu de `INTEGER` dans la colonne `Type`...).

```
mysql> DESC FilmSimple;
```

```
+------------+-------------+------+-----+---------+-------+
| Field      | Type        | Null | Key | Default | Extra |
+------------+-------------+------+-----+---------+-------+
| titre      | varchar(30) | YES  |     | NULL    |       |
| annee      | int(11)     | YES  |     | NULL    |       |
| nomMES     | varchar(30) | YES  |     | NULL    |       |
| prenomMES  | varchar(30) | YES  |     | NULL    |       |
| anneeNaiss | int(11)     | YES  |     | NULL    |       |
+------------+-------------+------+-----+---------+-------+
```

Pour des raisons obscures, MySQL affiche `int(11)` au lieu de `INTEGER` dans la colonne `Type`....

Pour détruire une table, on dispose de la commande `DROP TABLE`.

```
mysql> DROP TABLE FilmSimple;
Query OK, 0 rows affected (0.01 sec)
```

Il est assez fastidieux d'entrer au clavier toutes les commandes de *mysql*, d'autant que toute erreur de frappe implique de recommencer la saisie au début. Une meilleure solution est de créer un fichier contenant les commandes et de l'exécuter. Voici le fichier *FilmSimple.sql* (nous parlerons de *script SQL* à partir de maintenant).

Exemple 11 *FilmSimple.sql : Fichier de création de la table FilmSimple*

```
# Création d'une table 'FilmSimple'

USE Films;

CREATE TABLE FilmSimple
  (titre      VARCHAR (30),
   annee      INTEGER,
   nomMES     VARCHAR (30),
   prenomMES  VARCHAR (30),
   anneeNaiss INTEGER
  )
;
```

Un script SQL peut contenir tout un ensemble de commandes, chacune devant se terminer par un ';'. Ici on s'assure d'abord que la base courante est bien *Films*, avec la commande `USE Films`, avant d'exécuter la commande `CREATE TABLE`. Toutes les lignes commençant par '#' sont des commentaires.

On indique à *mysql* qu'il doit prendre ses commandes dans ce fichier au lieu de l'entrée standard de la manière suivante :

```
% mysql < FilmSimple.sql
```

Le '<' permet une redirection de l'entrée standard (par défaut la console utilisateur) vers *FilmSimple.sql*.

Insertion de données

Nous avons maintenant une table *FilmSimple* dans laquelle nous pouvons insérer des données avec la commande SQL `INSERT`. Voici sa syntaxe :

```
INSERT INTO FilmSimple (titre, annee, nomMES, prenomMES)
VALUES ('Pulp Fiction', 1996, 'Quentin', 'Tarantino');
```

On indique donc la table dans laquelle on veut insérer une ligne, puis la liste des attributs auxquels ont va affecter une valeur. Les attributs qui n'apparaissent pas, comme, pour cet exemple l'année de naissance du metteur en scène `anneeNaiss`, auront une valeur dite `NULL` sur laquelle nous reviendrons plus tard.

La dernière partie de la commande `INSERT` est la liste des valeurs, précédée du mot-clé `VALUES`. Il doit y avoir autant de valeurs que d'attributs, et les chaînes de caractères doivent être encadrées par des apostrophes simples ('), ce qui permet d'y introduire des blancs.

Remarque : MySQL accepte les apostrophes doubles ("), mais ce n'est pas conforme à la norme SQL ANSI. Il est préférable de prendre l'habitude d'utiliser systématiquement les apostrophes simples.

Voici l'exécution de cette commande `INSERT` avec l'utilitaire *mysql*.

```
% mysql Films

Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 86 to server version: 3.23.9

Type 'help' for help.

mysql> INSERT INTO FilmSimple
    -> (titre, annee, nomMES, prenomMES)
    ->  VALUES ('Pulp Fiction', 1996,  'Quentin', 'Tarantino');
Query OK, 1 row affected (0.16 sec)
```

La commande `INSERT` est en fait rarement utilisée directement, car la syntaxe est assez lourde et surtout il n'y pas de contrôle sur les valeurs des attributs. Le plus souvent l'insertion de lignes dans une table se fait avec l'une des deux méthodes suivantes.

Saisie dans une interface graphique. Ce type d'interface offre une aide à la saisie, permet des contrôles, et facilite la tâche de l'utilisateur. Nous verrons comment créer de telles interfaces sous forme de formulaires HTML.

Chargement « en masse » à partir d'un fichier. Dans ce cas un programme lit les informations dans le fichier contenant les données, et effectue répétitivement des ordres `INSERT` pour chaque ligne trouvée.

MySQL fournit une commande assez puissante, `LOAD DATA`, qui évite dans beaucoup de cas d'avoir à écrire un programme spécifique pour charger des fichiers dans une base. Cette commande est capable de lire de nombreux formats différents. Nous reprenons le fichier *films.txt* que nous avions utilisé pour illustrer la programmation CGI (voir page 33) dont le contenu est rappelé ci-dessous

```
Alien 1979 Scott Ridley 1943
Vertigo 1958 Hitchcock Alfred 1899
Psychose 1960 Hitchcock Alfred 1899
Kagemusha 1980 Kurosawa Akira 1910
Volte-face 1997 Woo John 1946
Titanic 1997 Cameron James 1954
Sacrifice 1986 Tarkovski Andrei 1932
```

Voici comment on utilise la commande `LOAD DATA` pour insérer en une seule fois le contenu de *films.txt* dans la table *FilmSimple*.

```
LOAD DATA LOCAL INFILE 'films.txt'
INTO TABLE FilmSimple
FIELDS TERMINATED BY ' ';
```

Voici quelques explications sur les options utilisées :

- l'option `LOCAL` indique au serveur que le fichier se trouve sur la machine du client *mysql*. Par défaut, le serveur cherche le fichier sur sa propre machine, dans le répertoire contenant la base de données ;
- on donne ici simplement le nom '*films.txt*', *ce qui suppose que le client mysql a été lancé dans le répertoire où se trouve ce fichier*. Si ce n'est pas le cas il faut indiquer le chemin d'accès complet.
- enfin il existe de nombreuses options pour indiquer le format du fichier. Ici on indique qu'une ligne dans le fichier correspond à une ligne dans la table, et que les valeurs des attributs dans le fichier sont séparées par des blancs.

L'exécution sous *mysql* donne le résultat suivant :

```
% mysql
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 8 to server version: 3.23.9

Type 'help' for help.

mysql> LOAD DATA LOCAL INFILE  'films.txt'
    -> INTO TABLE  FilmSimple
    -> FIELDS TERMINATED BY ' ';
Query OK, 7 rows affected (0.00 sec)
Records: 7  Deleted: 0  Skipped: 0  Warnings: 0
```

Il s'agit d'un format simple mais notablement insuffisant, comme nous l'avons déjà souligné. Il n'est pas possible d'avoir des blancs dans le titre de films, ou d'ignorer la valeur d'un attribut. On ne saurait pas charger la description du film *Pulp Fiction* avec ce format.

Heureusement `LOAD DATA` sait traiter des formats de fichier beaucoup plus complexes. Une description complète de cette commande est donnée dans l'annexe B.

Interrogation et modification

Le langage SQL propose les quatre opérations essentielles de manipulation de données : *insertion*, *destruction*, *mise à jour* et *recherche*. Ces opérations sont communément désignées par le terme de *requêtes*. L'ensemble des commandes permettant d'effectuer ces opérations est le *Langage de Manipulation de Données*, ou LMD, par opposition au *Langage de Définition de Données* ou LDD.

Nous avons déjà vu la commande `INSERT` qui effectue des insertions. Nous introduisons maintenant les trois autres opérations en commençant par la recherche qui est de loin la plus complexe. Les exemples qui suivent s'appuient sur la table *FilmSimple* contenant les quelques lignes insérées précédemment.

```
SELECT     titre, annee
FROM       FilmSimple
WHERE      annee > 1980
```

Ce premier exemple nous montre la structure de base d'une recherche avec SQL, avec les trois clauses `SELECT`, `FROM` et `WHERE`.

- `FROM` indique la table dans laquelle on trouve les attributs utiles à la requête. Un attribut peut être « utile » de deux manières (non exclusives) : (1) on souhaite afficher son contenu (clause `SELECT`, (2) il a une valeur particulière (une constante ou la valeur d'un autre attribut) que l'on indique dans la clause `WHERE`.
- `SELECT` indique la liste des attributs constituant le résultat.
- `WHERE` indique les conditions que doivent satisfaire les lignes de la base pour faire partie du résultat.

Remarque : sous Unix, une table comme *FilmSimple* est stockée par MySQL dans un fichier de nom *FilmSimple*. Comme Unix distingue les majuscules et les minuscules pour les noms de fichier, il faut absolument respecter la casse dans le nom des tables, sous peine d'obtenir le message `Table does not exist`.

Cette requête peut être directement effectuée sous *mysql*, ce qui donne le résultat suivant.

```
mysql> SELECT  titre, annee
    -> FROM    FilmSimple
    -> WHERE    annee > 1980;
+------------+-------+
| titre      | annee |
```

```
+------------+-------+
| Volte-face |  1997 |
| Titanic    |  1997 |
| Sacrifice  |  1986 |
+------------+-------+
3 rows in set (0.00 sec)
```

N'oubliez pas le point-virgule pour finir une commande. La requête SQL la plus simple est celle qui affiche toute la table, sans faire de sélection (donc sans clause `WHERE`) et en gardant tous les attributs. Dans un tel cas on peut simplement utiliser le caractère '*' pour désigner la liste de tous les attributs.

```
mysql> SELECT * FROM FilmSimple;
+------------+-------+-----------+-----------+------------+
| titre      | annee | nomMES    | prenomMES | anneeNaiss |
+------------+-------+-----------+-----------+------------+
| Alien      |  1979 | Scott     | Ridley    |       1943 |
| Vertigo    |  1958 | Hitchcock | Alfred    |       1899 |
| Psychose   |  1960 | Hitchcock | Alfred    |       1899 |
| Kagemusha  |  1980 | Kurosawa  | Akira     |       1910 |
| Volte-face |  1997 | Woo       | John      |       1946 |
| Titanic    |  1997 | Cameron   | James     |       1954 |
| Sacrifice  |  1986 | Tarkovski | Andrei    |       1932 |
+------------+-------+-----------+-----------+------------+
7 rows in set (0.00 sec)
```

Les requêtes les plus complexes que l'on puisse faire à ce stade sont celles qui sélectionnent des films selon des critères comme « Les films dont le titre est *Vertigo*, ou dont le prénom du metteur en scène est *John*, ou qui sont parus dans les années 90 ». La clause `WHERE` permet la combinaison de ces critères avec les connecteurs `AND`, `OR` et l'utilisation éventuelle des parenthèses pour lever les ambiguïtés.

```
mysql> SELECT titre, annee
    -> FROM  FilmSimple
    -> WHERE titre = 'Vertigo'
    -> OR    prenomMES = 'Alfred'
    -> OR     (annee >= 1990 AND annee < 2000);
+------------+-------+
| titre      | annee |
+------------+-------+
| Vertigo    |  1958 |
| Psychose   |  1960 |
| Volte-face |  1997 |
| Titanic    |  1997 |
+------------+-------+
4 rows in set (0.00 sec)
```

Tant qu'il n'y a qu'une table à interroger, l'utilisation de SQL s'avère extrêmement simple. Le serveur fait tout le travail pour nous : accéder au fichier, lire les lignes, retenir celles qui satisfont les critères, satisfaire simultanément (ou presque) les demandes de plusieurs utilisateurs, etc. Dès qu'on interroge une base avec plusieurs tables, ce qui est la situation normale, les requêtes SQL deviennent un peu plus complexe.

Remarque : Comme pour PHP, nous introduisons SQL au fur et à mesure. Les requêtes sur plusieurs tables (jointures) sont présentées dans le chapitre 4.1, page 133. Les requêtes d'agrégation sont présentées dans ce même chapitre, page 147. Enfin le chapitre 7 propose un récapitulatif complet sur le langage.

Les commandes de mise à jour et de destruction sont des variantes du `SELECT`. On utilise la même clause `WHERE`, en remplaçant dans un cas le `SELECT` par `UPDATE`, et dans l'autre par `DELETE`. Voici

deux exemples.

– *Détruire tous les films antérieurs à 1960.*
  Le critère de sélection des films à détruire est exprimé par une clause `WHERE`.

  ```
  DELETE FROM FilmSimple WHERE annee <= 1960
  ```

  Les données détruites sont *vraiment* perdues. Ceux qui auraient l'habitude d'un système gérant les transactions doivent garder en mémoire qu'il n'y a pas de possibilité de retour en arrière avec `rollback` dans MySQL.
– *Changer le nom de 'John Woo' en 'Jusen Wu'.*
  La commande est légèrement plus complexe. On indique par une suite de `SET` *`attribut=valeur`* l'affectation de nouvelles valeurs à certains attributs des lignes modifiées.

  ```
  UPDATE FilmSimple SET nomMES='Wu', prenomMES='Yusen'
  WHERE nomMES = 'Woo'
  ```

  Même remarque que précédemment : toutes les lignes sont modifiées sans possibilité d'annulation. Une manière de s'assurer que la partie de la table affectée par un ordre `DELETE` ou `UPDATE` est bien celle que l'on vise est d'effectuer au préalable la requête avec `SELECT` et la même clause `WHERE`.

Voici l'exécution sous *mysql*.

```
mysql> DELETE FROM FilmSimple WHERE annee <= 1960;
Query OK, 2 rows affected (0.01 sec)
mysql>
mysql>  UPDATE FilmSimple SET nomMES='Wu', prenomMES='Yusen'
    ->  WHERE nomMES = 'Woo';
Query OK, 1 row affected (0.00 sec)
Rows matched: 1  Changed: 1  Warnings: 0
```

Quelques commandes utiles

Enfin *mysql* fournit tout un ensemble de commandes pour inspecter les tables, donner la liste des tables d'une base de données, etc. Voici une sélection des commandes les plus utiles. L'annexe B donne une liste exhaustive de toutes les fonctionnalités de MySQL.

– `SELECT DATABASE();` C'est une pseudo-requête SQL (il n'y a pas de `FROM`) qui affiche le nom de la base courante.
– `SELECT USER();` Idem, cette pseudo-requête affiche le nom de l'utilisateur courant.
– `SHOW DATABASES;` Affiche la liste des bases de données.
– `SHOW TABLES;` Affiche la liste des tables de la base courante.
– `SHOW COLUMNS FROM` *`NomTable`*`;` Affiche la description de *`NomTable`*.

3.2.3 L'interface phpMyAdmin

phpMyAdmin est un outil entièrement écrit en PHP qui fournit une interface simple et très complète pour administrer une base MySQL. La plupart des commandes de l'utilitaire *mysql* peuvent s'effectuer par l'intermédiaire de phpMyAdmin, les opérations possibles dépendant bien sûr des droits de l'utilisateur qui se connecte à la base. Voici une liste des principales possibilités :

1. Créer et détruire des bases de données (sous le compte `root` de MySQL).
2. Créer, détruire, modifier la description des tables.
3. Consulter le contenu des tables, modifier certaines lignes ou les détruire, etc.
4. Exécuter des requêtes SQL interactivement.

5. Charger des fichiers dans des tables et, réciproquement, récupérer le contenu de tables dans des fichiers ASCII.

Beaucoup de fournisseurs d'accès utilisent ce module pour permettre la création, modification ou mise à jour d'une base de données personnelle à distance, à l'aide d'un simple navigateur. L'annexe A décrit l'installation de phpMyAdmin. Même s'il ne dispense pas complètement de l'utilisation de l'utilitaire *mysql*, il permet de faire beaucoup d'opérations simples de manière conviviale.

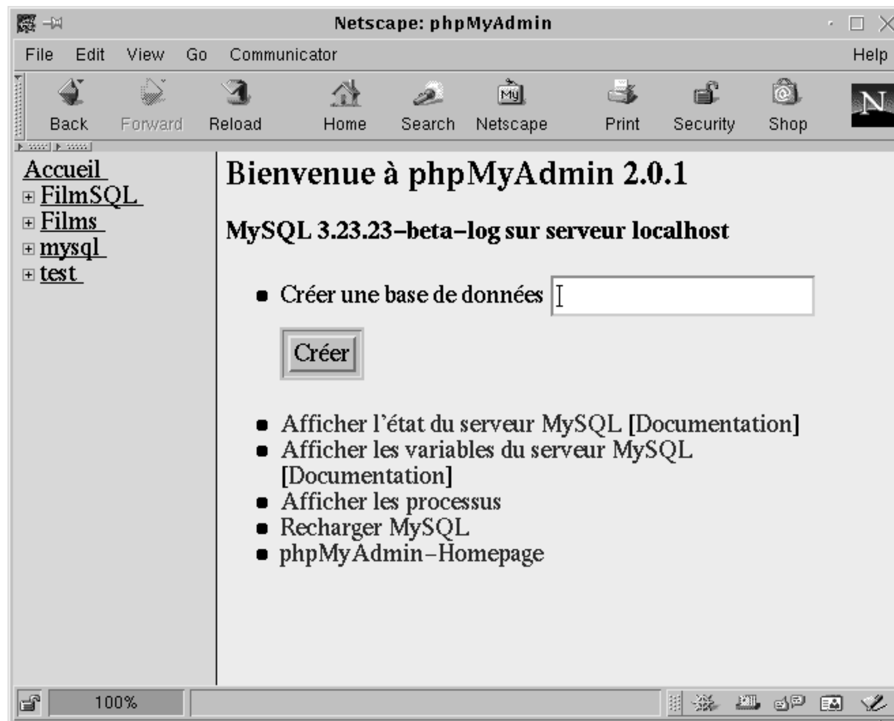

FIG. 3.3 – *La page d'accueil de phpMyAdmin*

La figure 3.3 montre une copie d'écran de la page d'accueil de phpMyAdmin. L'écran est divisé en deux parties. Sur la gauche on voit toutes les bases de données gérées par le serveur (si vous accédez au système d'un fournisseur d'accès, vous ne verrez certainement que votre base personnelle). Nous avons ici, en partant du bas, la base *test* automatiquement créée par MySQL pour faire des essais, la base *mysql* qui contient les informations sur les utilisateurs (voir chapitre 8), la base *Films* que nous allons utiliser tout au long de ce livre, enfin une base simplifiée, *FilmSQL*, qui nous servira pour notre présentation de SQL dans le chapitre 7.

Cette partie gauche reste affichée en permanence. La partie droite présente l'ensemble des opérations disponibles en fonction du contexte. Initialement, si le compte de connexion utilisé est `root`, phpMyAdmin propose de consulter la situation du serveur et des clients MySQL.

En cliquant sur une des bases, on obtient la liste des tables, et toute une liste d'actions à effectuer sur cette base. La figure 3.4 montre cette seconde page (noter qu'il s'agit d'un formulaire HTML).

Liste des tables Le haut de la page est consacré à la liste des tables. Ici on ne voit que la table *FilmSimple* que nous venons de créer avec *mysql*. Sur cette table on peut effectuer les opérations suivantes.

1. *Afficher* donne le contenu de la table.
2. *Sélectionner* propose un petit formulaire permettant de sélectionner une partie de la table.
3. *Insérer* présente un autre formulaire, créé dynamiquement par phpMyAdmin, cette fois pour insérer des données dans la table.

4. *Propriétés* donne la description de la table. Cette option donne accès à une autre page, assez complète, qui permet de modifier la table en ajoutant ou en supprimant des attributs.

   C'est là aussi que l'on peut échanger des données entre une table et un fichier ASCII. Une option *Insérer un fichier texte dans la table* vous permet de transmettre un fichier (par exemple *films.txt*) que phpMyAdmin utilisera dans une commande `LOAD DATA INFILE`.
5. *Effacer* détruit la table (phpMyAdmin demande confirmation).
6. *Vide* détruit toutes les lignes.

Exécution de commande La fenêtre placée en dessous de la liste des tables permet d'entrer des commandes SQL directement.

Pour créer la table *FilmSimple*, on peut copier/coller directement la commande `CREATE TABLE` dans cette fenêtre et l'exécuter. De même on peut effectuer des `INSERT`, des `SELECT`, et toutes les commandes que nous avons vues dans la section précédente. Cette fenêtre est, dans phpMyAdmin, la fonctionnalité la plus proche de l'utilitaire *mysql*.

Requête par un exemple Ce lien donne accès à un formulaire aidant à la construction de requêtes SQL complexes, sans connaître SQL.

Schéma de la base Cette partie permet de créer un fichier contenant toutes les commandes de création de la base, ainsi que, optionnellement, les ordres d'insertion des données sous forme de commandes `INSERT`. En d'autres termes vous pouvez faire une sauvegarde complète, sous forme d'un fichier ASCII. En choisissant l'option *transmettre*, le fichier est transmis au navigateur.

Création d'une nouvelle table propose un formulaire pour créer une nouvelle table. Avant le bouton « Exécuter », il faut entrer le nom de la table et le nombre d'attributs.

L'utilisation de phpMyAdmin est simple et s'apprend en pratiquant. Bien que cet outil, en offrant une interface de saisie, économise beaucoup de frappe au clavier, il s'avère quand même nécessaire à l'usage de connaître les commandes SQL pour comprendre les actions effectuées et les différentes options possibles. Dans tout ce qui suit, nous continuerons à utiliser l'outil *mysql* pour présenter les commandes du langage SQL, sachant que l'équivalent existe sous phpMyAdmin.

Si vous voulez comprendre de manière approfondie comment cet outil a été réalisé, vous pouvez consulter directement le code PHP de phpMyAdmin, librement disponible. Il présente beaucoup d'aspects intéressants, en particulier la gestion des langues, l'outil d'aide à la création de requête, et le module permettant de générer dynamiquement les commandes de création d'une base et de son contenu. La compréhension du code demande quand même une certaine pratique : attendez d'avoir acquis une bonne connaissance de PHP.

3.3 Accès à MySQL avec PHP

Maintenant que nous disposons d'une base MySQL, nous pouvons aborder les outils d'accès à cette base à partir de scripts PHP. Nous étudions successivement dans cette section les aspects suivants :

L'interface fonctionnelle MySQL/PHP. Il s'agit d'un ensemble de fonctions qui, pour l'essentiel, permettent de se connecter à MySQL, d'exécuter des requêtes SQL et de récupérer le résultat que l'on peut ensuite afficher dans un page HTML.

Interrogation à partir de formulaires HTML. Nous reprenons l'exemple donné sous la forme d'un programme CGI dans le chapitre 2 et montrons son équivalent avec PHP. Beaucoup plus simple !

Insertions et mises à jour. Toujours à partir de formulaires HTML, on peut créer des scripts PHP qui insèrent de nouvelles informations ou modifient celles qui existent déjà.

Enfin nous montrons, en reprenant le code produit au cours de ces étapes et en le réorganisant, comment l'utilisation de fonctions permet d'obtenir des scripts plus lisibles et concis.

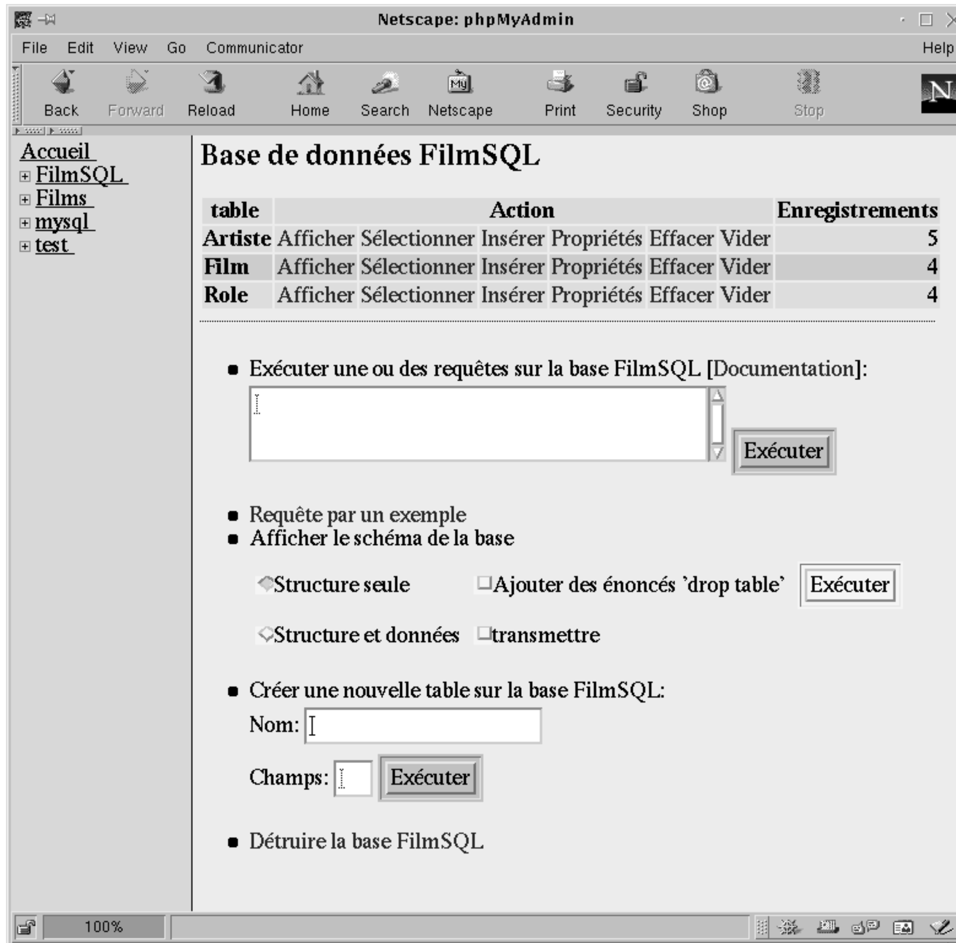

FIG. 3.4 – *Actions sur une base avec phpMyAdmin*

3.3.1 L'interface MySQL/PHP

PHP communique avec MySQL par l'intermédiaire d'un ensemble de fonctions qui permettent de récupérer, modifier, ou créer à peu près toutes les informations relatives à une base de données. Parmi ces informations, il faut compter bien entendu le contenu des tables, mais également leur description (le *schéma* de la base). L'utilitaire phpMyAdmin utilise par exemple les fonctions permettant d'obtenir le schéma pour présenter une interface d'administration, générer à la volée des formulaires de saisie, etc.

Le tableau 3.1 donne la liste des principales fonctions de l'API. Nous renvoyons à l'annexe D pour une liste exhaustive des fonctions MySQL/PHP.

Voici maintenant ces fonctions en action. Le script suivant effectue une recherche de toutes les lignes de la table *FilmSimple* et affiche la liste des films dans une page HTML.

Exemple 12 *ExMyPHP1.php* : *Accès à MySQL avec PHP*

```
<HTML><HEAD>
<TITLE>Connexion à MySQL</TITLE>
<LINK REL=stylesheet HREF="films.css" TYPE="text/css">
</HEAD>
<BODY>

<H1>Interrogation de la table FilmSimple</H1>

<?php
```

Fonction	Description
mysql_connect	Pour établir une connexion avec MySQL, pour un compte utilisateur, et un serveur donnés. Renvoie une valeur qui peut être utilisée ensuite pour dialoguer avec le serveur.
mysql_pconnect	Idem, mais avec une connexion *persistante* (voir annexe D). Cette deuxième version est plus performante.
mysql_select_db	Permet de se placer dans une base de données. C'est l'équivalent de la commande `USE base` sous *mysql*.
mysql_query	Pour exécuter une requête SQL. Renvoie une variable représentant le résultat de la requête.
mysql_fetch_object	Permet de récupérer une des lignes du résultat, et positionne le curseur sur la ligne suivante. La ligne est représentée sous forme d'un *objet* (un groupe de valeurs).
mysql_fetch_row	Permet de récupérer une des lignes du résultat, et positionne le curseur sur la ligne suivante. La ligne est représentée sous forme d'un *tableau* (une liste de valeurs).
mysql_error	Renvoie le message de la dernière erreur rencontrée.

TAB. 3.1 – *Principales fonctions de l'API MySQL/PHP*

```
  require ("Connect.php");

  $connexion = mysql_pconnect (SERVEUR, NOM, PASSE);

  if (!$connexion)
  {
    echo "Désolé, connexion à " . SERVEUR . " impossible\n";
    exit;
  }

  if (!mysql_select_db (BASE, $connexion))
  {
    echo "Désolé, accès à la base " . BASE . " impossible\n";
    exit;
  }

  $resultat = mysql_query ("SELECT * FROM FilmSimple", $connexion);

  if ($resultat)
  {
    while ($film = mysql_fetch_object ($resultat))
    {
       echo "$film->titre, paru en $film->annee, réalisé "
            . "par  $film->prenomMES $film->nomMES.<BR>\n";
    }
  }
  else
  {
    echo "<B>Erreur dans l'exécution de la requête.</B><BR>";
    echo "<B>Message de MySQL :</B> " .  mysql_error($connexion);
  }
?>
</BODY></HTML>
```

Nous allons commenter soigneusement ce code qui est représentatif d'une bonne partie des techniques

nécessaires pour accéder à une base MySQL et en extraire des informations. Le script proprement dit se réduit à la partie comprise entre les balises `<?php` et `?>`.

Inclusion de fichiers - Constantes

La première instruction est

```
require ("Connect.php");
```

La commande `require` permet d'inclure le contenu d'un fichier dans le script. Certaines informations sont communes à beaucoup de scripts, et les répéter systématiquement est à la fois une perte de temps et une grosse source d'ennui le jour où il faut effectuer une modification dans n versions dupliquées. Ici on a placé dans le fichier *Connect.php* quelques informations de base sur le site : le nom du serveur, le nom de la base et le compte d'accès à la base.

```
<?php
  define (NOM,"adminFilms");
  define (PASSE, "mdpAdminFilms");
  define (SERVEUR, "localhost");
  define (BASE, "Films");
?>
```

La commande `define` permet de définir des *constantes*, en fait des symboles qui correspondent à des valeurs qui ne peuvent être modifiées. L'utilisation systématique des constantes, définies en un seul endroit (un fichier que l'on peut insérer à la demande) garantit l'évolutivité du site. Si le serveur devient par exemple *magellan* et le nom de la base *Movies*, il suffira de faire la modification à un seul endroit. Accessoirement, l'utilisation de symboles simples permet de ne pas avoir à se souvenir de valeurs ou de textes qui peuvent être compliqués.

Remarque : Il est tentant de donner une extension autre que *.php* aux fichiers contenant les scripts. Le fichier *Connect.php* par exemple pourrait être nommé *Connect.inc* pour bien indiquer qu'il est destiné à être inclus. Attention cependant : il devient alors possible de consulter le contenu du fichier avec l'URL *http://serveur/Connect.inc*. L'extension *.inc* étantt inconnue du programme serveur, ce dernier choisira alors de transmettre le contenu en clair (en-tête `text/plain`) au client. Cela serait très regrettable dans le cas de *Connect.php* qui contient un mot de passe. Un fichier d'extension *.php* sera, lui, toujours soumis par le programme serveur au filtre de l'interpréteur PHP et son contenu n'est jamais visible par le client web.

Connexion au serveur

Donc nous disposons avec ce `require` des symboles de constantes `NOM`, `PASSE`, `BASE` et `SERVEUR` [1], soit tous les paramètres nécessaires à la connexion à MySQL.

```
$connexion = mysql_pconnect (SERVEUR, NOM, PASSE);
```

La fonction *mysql_pconnect* essaie d'établir une connexion avec le serveur *mysqld*. En cas de succès une valeur positive est renvoyée, qui doit ensuite être utilisée pour dialoguer avec le serveur. En cas d'échec *mysql_pconnect* affiche un message d'erreur et renvoie une valeur nulle.

Remarque : Si vous voulez éviter que MySQL envoie un message en cas d'échec à la connexion, vous pouvez préfixer le nom de la fonction par '`@`'. C'est à vous alors de tester si la connexion est établie et

1. L'utilisation des majuscules pour les constantes n'est pas une obligation, mais facilite la lecture.

d'afficher un message selon vos propres normes de présentation.

Avant de continuer, il faut vérifier que la connexion est bien établie. Pour cela on peut tester la valeur de la variable `$connexion`, et, le cas échéant, afficher un message et interrompre le script avec `exit`.

```
if (!$connexion)
{
  echo "Désolé, connexion à " . SERVEUR . " impossible";
  exit;
}
```

La commande `echo` écrit sur la sortie standard, laquelle est transmise par le serveur web au navigateur. En d'autres termes, tout ce qui est écrit avec `echo` (ou la fonction similaire `printf`) constitue le (sous-)document HTML résultat de l'exécution du script.

Avec PHP, toute valeur non nulle est considérée comme vraie, le 0 ou la chaîne vide étant interprétés comme faux. Donc au lieu d'effectuer un test de comparaison, on peut tester directement la valeur de la variable `$connexion`. Le test simple `if ($connexion)` donne un résultat inverse de `if ($connexion == 0)`.

En revanche, en inversant la valeur booléenne de `$connexion` avec l'opérateur de négation (!), on obtient un test équivalent, et la notation, très courante, `if (! $connexion)`. Donc le `if` est vérifié si `$connexion` est faux, ce qui est le but recherché.

Le même principe est appliqué au résultat de la fonction *mysql_select_db* qui renvoie, elle aussi, une valeur positive (donc vraie) si l'accès à la base réussit. D'où le test :

```
if (!mysql_select_db (BASE, $connexion))
```

Tous ces tests sont importants. Il peut y avoir beaucoup de raisons pour qu'un serveur ne soit plus disponible, ou qu'un compte de connexion ne soit plus valide, etc. Le fait de continuer le script, et donc d'effectuer des requêtes sans avoir de connexion, mène à des messages d'erreur assez désagréables. Bien entendu l'écriture systématique de tests et de messages alourdit le code : nous verrons comment écrire ce genre de chose une (seule) fois pour toutes en utilisant des fonctions.

Exécution de la requête

Le moment est enfin venu d'effectuer une requête ! On utilise la fonction *mysql_query*.

```
$resultat = mysql_query ("SELECT * FROM FilmSimple", $connexion);
```

Comme d'habitude, cette fonction renvoie une valeur positive si la fonction s'exécute correctement. En cas de problème, erreur de syntaxe par exemple, le bloc associé au `else` est exécuté. Il affiche le message fourni par MySQL *via* la fonction *mysql_error*.

```
echo "<B>Erreur dans l'exécution de la requête.</B><BR>";
echo "<B>Message de MySQL:</B>" . mysql_error();
```

Noter l'utilisation de balises HTML dans les chaînes de caractères, ainsi que l'utilisation de l'opérateur de concaténation de chaînes, '.'.

Affichage du résultat

Si la requête réussit, il ne reste plus qu'à récupérer le résultat. Ici nous avons à résoudre un problème classique d'interaction entre une base de données et un langage de programmation. Le résultat est un ensemble, arbitrairement grand, de lignes dans une table, et le langage ne dispose pas de structure pratique pour représenter cet ensemble. On peut penser à tout mettre dans un tableau à deux dimensions (c'est d'ailleurs possible avec PHP), mais se pose alors un problème d'occupation mémoire si le résultat est *vraiment* grand (plusieurs mégaoctets par exemple).

La technique habituellement utilisée est de parcourir les lignes une à une avec un *curseur* et d'appliquer le traitement à chaque ligne individuellement. Cela évite d'avoir à charger tout le résultat en même temps.

Ici on utilise une des fonctions *fetch* qui correspondent à l'implantation de cette notion de curseur dans MySQL.

```
$film = mysql_fetch_object ($resultat);
```

La fonction *mysql_fetch_object* prend une ligne dans le résultat (initialement on commence avec la première ligne) et positionne le curseur sur la ligne suivante. Donc à chaque appel on progresse d'une étape dans le parcours du résultat. Quand toutes les lignes ont été parcourues, la fonction renvoie 0.

Avec cette fonction, chaque ligne est renvoyée sous la forme d'un *objet*, que nous référençons avec la variable `$film` dans l'exemple. Nous aurons l'occasion de revenir sur ce concept, mais pour l'instant il suffit de considérer simplement qu'un objet est un groupe de valeurs, chacune étant identifiée par un nom d'attribut.

Dans notre cas ces noms sont naturellement les noms des attributs de la table *FilmSimple*. On accède à chaque attribut avec l'opérateur '->'. Donc `$film->titre` est le titre du film, `$film->nomMES` le nom du metteur en scène, etc.

L'affectation du résultat de *mysql_fetch_object* à la variable `$film` envoie elle-même une valeur, qui est 0 quand le résultat a été parcouru en totalité [2]. D'où la boucle d'affichage des films :

```
while ($film = mysql_fetch_object ($resultat))
{
   echo "$film->titre, paru en $film->annee, réalisé "
         . "par  $film->prenomMES $film->nomMES.<BR>\n";
}
```

On peut remarquer, dans l'instruction `echo` ci-dessus, l'introduction de variables directement dans les chaînes de caractères. Autre remarque importante : on utilise deux commandes de retour à la ligne, `<BR>` et `\n`. Elles n'ont pas du tout la même fonction, et il est instructif de réfléchir au rôle de chacune.

- la balise `<BR>` est incluse dans le document HTML produit. Elle indique au navigateur qu'un saut de ligne doit être effectué après la présentation de chaque film ;
- le caractère `\n` indique qu'un saut de ligne doit être effectuée *dans* le texte HTML, et pas dans la *présentation* du document qu'en fait le navigateur. Ce `\n` n'a en fait aucun effet sur cette présentation puisque nous avons vu que le formatage du texte HTML peut être quelconque. En revanche il permet de rendre ce texte, produit automatiquement, plus clair à lire si on doit y rechercher une erreur.

Voici le texte HTML produit par le script, tel qu'on peut le consulter avec la commande *View source* du navigateur. Si on n'avait pas mis un `\n`, tous les films seraient disposés sur une seule ligne !

Exemple 13 *ResMYPHP1.html* : *Résultat (texte HTML) du script*

```
<HTML><HEAD>
<TITLE>Connexion à MySQL</TITLE>
<LINK REL=stylesheet HREF="films.css" TYPE="text/css">
</HEAD>
<BODY>

<H1>Interrogation de la table FilmSimple</H1>

Alien, paru en 1979, réalisé par  Ridley Scott.<BR>
Vertigo, paru en 1958, réalisé par  Alfred Hitchcock.<BR>
Psychose, paru en 1960, réalisé par  Alfred Hitchcock.<BR>
Kagemusha, paru en 1980, réalisé par  Akira Kurosawa.<BR>
Volte-face, paru en 1997, réalisé par  John Woo.<BR>
Titanic, paru en 1997, réalisé par  James Cameron.<BR>
Sacrifice, paru en 1986, réalisé par  Andrei Tarkovski.<BR>
</BODY></HTML>
```

2. Voir l'annexe C et la partie sur les expressions pour plus d'explications.

3.3.2 Formulaires d'interrogation

Une des forces de PHP est son intégration naturelle avec les formulaires HTML. Plus besoin, contrairement à ce que nous avons vu dans la partie consacrée à la programmation CGI, de décrypter des chaînes de caractères au format complexe, puisque les champs du formulaire sont directement fournis sous forme de variable PHP. De plus, l'utilisation de SQL à la place d'un parcours laborieux du fichier nous donne des commandes à la fois beaucoup plus simples et plus puissantes.

Voici une variante du formulaire qui nous avait permis de déclencher un programme CGI écrit en C (exemple 8). Vous pouvez le tester directement sur notre site.

Exemple 14 *ExForm3.html* : *Le formulaire de l'exemple 8, avec script PHP*

```
<HTML><HEAD>
  <TITLE>Formulaire pour script PHP/MySQL</TITLE>
  <LINK REL=stylesheet HREF='films.css' TYPE='text/css'>
</HEAD><BODY>

<H1>Formulaire pour script PHP/MySQL</H1>

<FORM ACTION="ExMyPHP2.php" METHOD=POST>

Ce formulaire vous permet d'indiquer des paramètres pour
la recherche de films :
  <P>
  Titre :  <INPUT TYPE=TEXT SIZE=20 NAME = 'titre' VALUE='%'><BR>
  Le caractère '%' remplace n'importe quelle chaîne.
  <P>
  Année début : <INPUT TYPE=TEXT SIZE=4 NAME='anMin' VALUE=1900>
  Année fin : <INPUT TYPE=TEXT SIZE=4 NAME='anMax' VALUE=2100> <P>

  <B>Comment combiner ces critères. </B>
  ET <INPUT TYPE=RADIO NAME='comb' VALUE='ET' CHECKED>
  OU <INPUT TYPE=RADIO NAME='comb' VALUE='OU'> ?
  <P>
  <INPUT TYPE=SUBMIT VALUE='Rechercher'>
</FORM>

</BODY></HTML>
```

Il y a deux différences notables.

1. au lieu d'appeler un programme CGI, l'attribut `ACTION` fait référence à un script PHP ;
2. moins visible, mais tout aussi important : on peut maintenant entrer dans le champ `titre` non seulement un titre de film complet, mais aussi des titres partiels, complétés par le caractère '%' qui signifie, pour SQL, une chaîne quelconque.

Donc on peut, par exemple, rechercher tous les films commençant par 'Ver' en entrant 'Ver%', ou tous les films contenant un caractère blanc avec '% %'. Le fichier ci-dessous est le script PHP associé au formulaire précédent (à comparer avec le programme C du chapitre 2 !). Pour plus de concision, nous avons omis tous les tests portant sur la connexion et l'exécution de la requête SQL qui peuvent – devraient – être repris comme dans l'exemple 12.

Exemple 15 *ExMyPHP2.php* : *Le script associé au formulaire de l'exemple 14*

```
<HTML><HEAD>
<TITLE>Résultat de l'interrogation par formulaire</TITLE>
<LINK REL=stylesheet HREF="films.css" TYPE="text/css">
</HEAD>
<BODY>
```

```
<H1>Résultat de l'interrogation par formulaire</H1>

<?php
  require ("Connect.php");

  echo "<B>Titre = $titre anMin = $anMin anMax=$anMax\n";
  echo "Combinaison logique : $comb<P></B>\n";

  if ($comb == 'ET')
    $requete = "SELECT * FROM FilmSimple "
          . "WHERE titre LIKE '$titre' "
          . "AND annee BETWEEN $anMin and $anMax";
  else
    $requete = "SELECT * FROM FilmSimple "
          . "WHERE titre LIKE '$titre' "
          . "OR (annee BETWEEN $anMin and $anMax)";

  $connexion = mysql_pconnect (SERVEUR, NOM, PASSE);
  mysql_select_db (BASE, $connexion);
  $resultat = mysql_query ($requete, $connexion);
  while ( ($film = mysql_fetch_object ($resultat)))
       echo "$film->titre, paru en $film->annee, réalisé "
            . "par  $film->prenomMES $film->nomMES.<BR>\n";
?>
</BODY></HTML>
```

Les variables `$titre`, `$anMin`, `$anMax` et `$comb` sont définies dès l'entrée du script. En testant la valeur de `$comb`, qui peut être soit 'ET', soit 'OU', on détermine quel est l'ordre SQL à effectuer. Cet ordre utilise deux comparateurs, `LIKE` et `BETWEEN`. `LIKE` est un opérateur de *pattern matching* : il renvoie vrai si la variable `$titre` de PHP peut être rendue égale à l'attribut `titre` en remplaçant dans `$titre` le caractère '%' par n'importe quelle chaîne.

La requête SQL est placée dans une chaîne de caractères qui est ensuite exécutée.

```
$requete = "SELECT * FROM FilmSimple "
           . "WHERE titre LIKE '$titre' "
           . "AND annee BETWEEN $anMin and $anMax";
```

Dans l'instruction ci-dessus, on utilise la concaténation de chaînes ('.') pour disposer de manière plus lisible les différentes parties de la requête. On exploite ensuite la capacité de PHP à reconnaître l'insertion de variables dans une chaîne (grâce au préfixe $) et à remplacer ces variables par leur valeur. En supposant que l'on a saisi `Vertigo`, 1980 et 2000 dans ces trois champs, la variable `$requete` sera la chaîne de caractères suivante :

```
SELECT * FROM FilmSimple WHERE titre LIKE 'Vertigo'
AND annee BETWEEN 1980 AND 2000
```

Il faut toujours encadrer une chaîne de caractères comme `$titre` par des apostrophes simples (') car MySQL ne saurait pas faire la différence entre `Vertigo` et un attribut de la table. De plus cette chaîne de caractères peut éventuellement contenir des blancs, ce qui poserait des problèmes. Les apostrophes simples sont acceptés au sein d'une chaîne encadrée par des apostrophes doubles, et réciproquement.

Remarque : Que se passe-t-il si le titre du film contient lui même des apostrophes simples, comme, par exemple, `L'affiche rouge` ? Et bien PHP préfixe par \, pour les données provenant de formulaires, tous les caractères qui peuvent poser problème. La chaîne transmise sera donc `L\'affiche rouge`, et MySQL interprétera correctement ce guillemet comme faisant partie de la chaîne et pas comme un délimiteur. Ce comportement de PHP est activé par l'option `magic_quote_gpc` qui devrait normalement

être à on dans le fichier de configuration *php.ini* (voir annexe A). Sinon les fonctions *addSlashes* et *stripSlashes* permettent d'ajouter ou des supprimer les caractères d'échappement dans une chaîne de caractères.

En revanche, les apostrophes sont inutiles pour les numériques comme `$anMin` et `$anMax` qui ne peuvent être confondus avec des noms d'attribut et ne soulèvent donc pas de problème d'interprétation. Il faut quand même noter que nous ne faisons aucun contrôle sur les valeurs saisies, et qu'un utilisateur malicieux qui place des caractères dans les dates, ou transmet des chaînes vides, causera quelques soucis à ce script (vous pouvez d'ailleurs essayer, sur notre site).

Une dernière remarque : ce script PHP est associé au formulaire 14 puisqu'il attend des paramètres que le formulaire a justement pour objectif de collecter et transmettre. Cette association est cependant assez souple pour que tout autre moyen de passer des paramètres au script soit acceptée. Par exemple l'introduction des valeurs dans l'URL, sous la forme ci-dessous, est tout à fait valable.

```
ExMyPHP2.php?titre=Vert%&anMin=1980&anMax=2000&comb=OR
```

Pour tout script, on peut donc envisager de se passer du formulaire en utilisant la méthode ci-dessus, plus laborieuse, mais aussi parfois plus souple. Il est possible par exemple de récupérer la description (sous forme HTML) du film *Vertigo* avec l'URL *http://us.imdb.com/Title?Vertigo*, qui fait directement appel à un programme CGI du site *imdb.com*. Cela signifie que l'on n'a pas de garantie sur la provenance des données soumises à un script, et qu'elles n'ont pas forcément été soumises aux contrôles du formulaire prévu pour être associé à ce script.

3.3.3 Formulaires de mises à jour

L'interaction avec un site comprenant une base de données implique la possibilité d'effectuer des mises à jour sur cette base. Un exemple très courant est *l'inscription* d'un visiteur afin de lui accorder un droit d'utilisation du site. Là encore les formulaires constituent la méthode normale de saisie des valeurs à placer dans la base.

Nous donnons ci-dessous l'exemple de la combinaison d'un formulaire et d'un script PHP pour effectuer des insertions, modifications ou destructions dans la base de données des films. Cet exemple est l'occasion d'étudier quelques techniques avancées de définition de tables avec MySQL et de compléter le passage des paramètres entre le formulaire et PHP.

Une table plus complète

L'exemple utilise une version plus complète de la table stockant les films.

Exemple 16 *FilmComplet.sql : Fichier de création de FilmComplet*

```
# Création d'une table 'FilmComplet'

USE Films;

CREATE TABLE FilmComplet
  (titre      VARCHAR (30),
   annee      INTEGER,
   nomMES     VARCHAR (30),
   prenomMES  VARCHAR (30),
   anneeNaissance INTEGER,
   pays   ENUM ("FR", "US", "DE", "JP"),
   genre  SET ("C", "D", "H", "S")
   resume       TEXT,
  )
;
```

La table `FilmComplet` comprend quelques nouveaux attributs, dont trois utilisent des types de données particuliers.

1. l'attribut `pays` est un type *énuméré* : la valeur – unique – que peut prendre cet attribut doit appartenir à un ensemble donné explicitement au moment de la création de la table avec le type `ENUM` ;
2. l'attribut `genre` est un type *ensemble* : il peut prendre une *ou plusieurs* valeurs parmi celle qui sont énumérées avec le type `SET` ;
3. enfin l'attribut `resume` est une *longue* chaîne de caractères de type `TEXT` dont la taille peut aller jusqu'à 65 535 caractères.

Ces trois types de données ne font pas partie de la norme SQL. En particulier, une des règles de base dans un SGBD relationnel est qu'un attribut, pour une ligne donnée, ne peut prendre plus d'une seule valeur. Le type `SET` de MySQL permet de s'affranchir – partiellement – de cette contrainte. On a donc décidé ici qu'un film pouvait appartenir à plusieurs genres.

Le formulaire

Le formulaire permettant d'effectuer des mises à jour sur la base (en fait seulement sur la table *Film-Complet*) est donné ci-dessous.

Exemple 17 *ExForm4.html* : *Formulaire de mise à jour*

```
<HTML><HEAD>
<TITLE>Formulaire complet</TITLE>
<LINK REL=stylesheet HREF="films.css" TYPE="text/css">
</HEAD>
<BODY>

 <FORM  ACTION="ExMyPHP3.php"  METHOD=POST>

  Titre : <INPUT TYPE=TEXT SIZE=20 NAME="titre"><BR>
  Année : <INPUT TYPE=TEXT SIZE=4 MAXLENGTH=4 NAME="annee" VALUE="2000">
  <P>
  Comédie   : <INPUT TYPE=CHECKBOX NAME='genre[]' VALUE='C'>
  Drame     : <INPUT TYPE=CHECKBOX NAME='genre[]' VALUE='D'>
  Histoire  : <INPUT TYPE=CHECKBOX NAME='genre[]' VALUE='H'>
  Suspense  : <INPUT TYPE=CHECKBOX NAME='genre[]' VALUE='S'>
  <P>
   France     : <INPUT TYPE=RADIO NAME='pays' VALUE='FR' CHECKED>
   Etats-Unis : <INPUT TYPE=RADIO NAME='pays' VALUE='US'>
   Allemagne  : <INPUT TYPE=RADIO NAME='pays' VALUE='DE'>
   Japon      : <INPUT TYPE=RADIO NAME='pays' VALUE='JP'>
  <P>

  Metteur en scène (prénom - nom) :
    <INPUT TYPE=TEXT SIZE=20 NAME="prenom">
    <INPUT TYPE=TEXT SIZE=20 NAME="nom"> <BR>

  Année de naissance : <INPUT TYPE=TEXT SIZE=4 MAXLENGTH=4
             NAME="anneeNaissance" VALUE=2000>
  <P>
  Résumé :  <TEXTAREA NAME='resume' COLS=30 ROWS=3>Résumé du film
            </TEXTAREA>

  <H1>Votre choix</H1>
  <INPUT TYPE=SUBMIT VALUE='Insérer' NAME='inserer' >
  <INPUT TYPE=SUBMIT VALUE='Modifier' NAME='modifier' >
  <INPUT TYPE=SUBMIT VALUE='Détruire' NAME='detruire' >
  <INPUT TYPE=RESET VALUE='Annuler'>
```

```
</FORM>

</BODY></HTML>
```

Il est assez proche de celui de l'exemple 6, avec quelques différences notables. Tout d'abord le nom du champ `genre` est `genre[]`.

```
Comédie    : <INPUT TYPE=CHECKBOX NAME='genre[]' VALUE='C'>
Drame      : <INPUT TYPE=CHECKBOX NAME='genre[]' VALUE='D'>
Histoire   : <INPUT TYPE=CHECKBOX NAME='genre[]' VALUE='H'>
Suspense   : <INPUT TYPE=CHECKBOX NAME='genre[]' VALUE='S'>
```

Pour comprendre l'utilité de cette notation, il faut se souvenir que les paramètres issus du formulaire sont passés au script sur le serveur sous la forme de paires *nom=valeur*. Ici on utilise un champ `CHECKBOX` puisqu'on peut affecter plusieurs genres à un film. Si on clique sur au moins deux des valeurs proposées, par exemple « Histoire » et « Suspense », la chaîne transmise au serveur aura la forme suivante :
`...&genre[]=H&genre[]=S&...`
Pour le script PHP exécuté par le serveur, cela correspond aux deux instructions suivantes :

```
  $genre[] = 'H';
  $genre[] = 'S';
```

Imaginons un instant que l'on utilise un nom de variable `$genre`, sans les crochets []. Alors pour PHP la deuxième affectation viendrait annuler la première et `$genre` n'aurait qu'une seule valeur, 'S'. La notation avec crochets indique que `$genre` est en fait un *tableau*, donc une liste de valeurs. Mieux : PHP incrémente automatiquement l'indice pour chaque nouvelle valeur placée dans un tableau. Les deux instructions ci-dessus créent un tableau avec deux entrées, indicées respectivement par 0 et 1, et stockant les deux valeurs 'H' et 'S'.

Une autre particularité du formulaire est l'utilisation de plusieurs boutons `SUBMIT`, chacun associé à un nom différent.

```
<INPUT TYPE=SUBMIT VALUE='Insérer' NAME='inserer' >
<INPUT TYPE=SUBMIT VALUE='Modifier' NAME='modifier' >
<INPUT TYPE=SUBMIT VALUE='Détruire' NAME='detruire' >
```

Quand l'utilisateur clique sur un des boutons, une seule variable PHP est créée, dont le nom correspond à celui du bouton utilisé. Le script peut tirer parti de ce mécanisme pour déterminer le type d'action à effectuer.

Le script PHP

La troisième composante de cette petite application de mise à jour est le script PHP.

Exemple 18 *ExMyPHP3.php* : *Le script de mise à jour de FilmComplet*

```
<HTML><HEAD>
<TITLE>Résultat de la mise-à-jour par formulaire</TITLE>
<LINK REL=stylesheet HREF="films.css" TYPE="text/css">
</HEAD>
<BODY>

<H1>Résultat de la mise-à-jour par formulaire</H1>

<?php
  require ("Connect.php");

  // Test du type de mise à jour effectuée
```

```
  echo "<HR><H2>\n";
  if (isset($inserer))      echo "Insertion du film $titre";
  elseif (isset($modifier)) echo "Modification du film $titre";
  elseif (isset($detruire)) echo "Destruction du film $titre";
  echo "</H2><HR>\n";

  // Affichage des données du formulaire

  echo "Titre: $titre <BR> annee: $annee<BR>Pays: $pays<BR>\n";
  for ($i=0; $i < count ($genre); $i++)
  {
     $chaineGenre .= $separateur . $genre[$i];
     $separateur = ",";
  }
  echo "Genres  = $chaineGenre<BR>";
  echo "Résumé  = $resume<P>\n";
  echo "Mis en scène par $prenom $nom\n";

  //  Connexion à la base, et création de l'ordre SQL

  $connexion = mysql_pconnect (SERVEUR, NOM, PASSE);
  mysql_select_db (BASE, $connexion);

  if (isset($inserer))
    $requete = "INSERT INTO FilmComplet (titre, annee, "
          . "prenomMES, nomMES, anneeNaissance,  pays, "
          . "genre, resume) VALUES ('$titre', $annee, "
          . "'$nom', '$prenom', $anneeNaissance, "
          . " '$pays', '$chaineGenre', '$resume') ";
  if (isset($modifier))
    $requete = "UPDATE FilmComplet SET annee=$annee, "
          . "prenomMES = '$prenom', nomMES='$nom', "
          . "anneeNaissance=$anneeNaissance, pays='$pays', "
          . "genre = '$chaineGenre', resume='$resume' "
          . " WHERE titre = '$titre' ";
  if (isset($detruire))
    $requete = "DELETE FROM FilmComplet WHERE  titre = '$titre'";

  // Exécution de l'ordre SQL

  $resultat = mysql_query ($requete, $connexion);
  echo "<HR>La requête $requete a été effectuée.\n";
?>
</BODY></HTML>
```

Ce script procède en plusieurs étapes, chacune donnant lieu à une insertion dans la page HTML qui est fournie en retour au serveur.

Tout d'abord on regarde quelle est la variable définie, soit `$inserer`, soit `$modifier`, soit `$detruire`, et on en déduit le type de mise à jour effectué. On l'affiche alors pour informer l'utilisateur que sa demande a été prise en compte. Comme indiqué précédemment, la variable `$inserer` est définie seulement si le bouton correspondant a été utilisé. On utilise la fonction *isset* de PHP pour tester l'existence d'une variable.

```
if (isset($inserer))      echo "Insertion du film $titre";
elseif (isset($modifier)) echo "Modification du film $titre";
elseif (isset($detruire)) echo "Destruction du film $titre";
```

La construction `if-elseif` permet de contrôler successivement les différentes valeurs possibles. On pourrait aussi utiliser une structure `switch`, ce qui permettrait en outre de réagir au cas où aucune des variables ci-dessus n'est définie.

On affiche ensuite les valeurs provenant du formulaire. La variable `$genre` est traitée de manière particulière.

```
for ($i=0; $i < count ($genre); $i++)
{
   $chaineGenre .= $separateur . $genre[$i];
   $separateur = ",";
}
echo "Genres  = $chaineGenre<BR>";
```

Rappelons que `$genre` est un tableau, dont chaque élément correspond à un des choix de l'utilisateur. La fonction *count* permet de connaître le nombre d'éléments, puis la boucle `for` est utilisée pour parcourir un à un ces éléments.

Au passage, on crée la variable `$chaineGenre`, un chaîne de caractères qui contient la liste des codes de genres, séparés par des virgules, selon le format attendu par MySQL. Si, par exemple, on a choisi « Histoire » et « Suspense » `$chaineGenre` contiendra "H,S".

Enfin on construit la requête `INSERT`, `UPDATE` ou `DELETE` selon le cas.

Discussion

Le script précédent a beaucoup de défauts qui le rendent impropre à une véritable utilisation. Une première catégorie de problèmes découle de la conception de la base de données elle-même. Il est par exemple possible d'insérer plusieurs fois le même film, une mise à jour peut affecter plusieurs films, il faut indiquer à chaque saisie l'année de naissance du metteur en scène même s'il figure déjà dans la base, etc. Nous décrirons dans le chapitre 4 une conception plus rigoureuse qui permet d'éviter ces problèmes.

Si on se limite à la combinaison HTML/PHP en laissant pour l'instant de côté la base MySQL, les faiblesses du script sont de deux natures.

Pas de contrôles. Aucun test n'est effectué sur les valeurs des données, et en particulier des chaînes vides peuvent être transmises pour tous les champs. De plus la connexion à MySQL, et l'exécution des requêtes, peuvent échouer pour des quantités de raisons : cet échec éventuel devrait être contrôlé.

Une ergonomie rudimentaire. En se restreignant à des scripts très simples, on a limité du même coup la qualité de l'interface. Par exemple on aimerait que les formulaires soient présentés avec un alignement correct des champs. Plus important : il serait souhaitable, au moment de mettre à jour les informations d'un film, de disposer comme valeur par défaut des valeurs déjà saisies

Ces problèmes peuvent se résoudre en ajoutant du code PHP pour effectuer des contrôles, ou en complexifiant la partie HTML. Il est clair que l'on aboutit rapidement à des scripts très longs et difficilement lisibles. Il devient alors indispensable de recourir à une structuration du code permettant de répartir les tâches en unités indépendantes.