

SQL Premiers pas avec MariaDB

Support de cours Version 1.5

Sommaire

1 Introduction.....	5
1.1 Définition.....	5
1.2 L'outillage.....	6
1.3 Pour les lecteurs pressés.....	8
2 Introduction à SQL.....	9
2.1 Création d'une base de test MariaDB.....	9
2.2 Création d'une table pivot.....	12
2.3 Quelques outils SQL à connaître.....	15
2.3.1 Netbeans.....	16
2.3.2 Clients SQL alternatifs.....	18
2.3.3 Outils de modélisation.....	18
2.3.4 Commandes MariaDB utiles.....	21
2.3.5 Outil pour la création de jeux de données.....	23
3 Les bases de SQL.....	25
3.1 Créer un jeu de données métier.....	25
3.2 Requêtes de type SELECT simples.....	27
3.3 Approfondissement sur les sous-requêtes.....	33
3.4 Petite introduction aux jointures.....	36
3.5 Petite introduction aux unions.....	40
3.6 Liste des fonctions disponibles.....	41
3.6.1 Les fonctions associées aux chaînes de caractère.....	41
3.6.2 Les fonctions associées aux dates.....	42
3.6.3 Les fonctions d'agrégation.....	44
3.6.3 Les fonctions associées aux nombres.....	47
3.7 Les requêtes d'insertion, de mise à jour et de suppression.....	48
3.7.1 Les requêtes INSERT.....	48
3.7.2 Les requêtes UPDATE.....	49
3.7.3 Les requêtes DELETE.....	50
3.8 Les astuces des pros.....	52

3.8.1 Identifier les doublons et les éliminer.....	52
3.8.2 Identifier les orphelins et les éliminer.....	54
3.8.3 Faire pivoter des données.....	55
3.8.4 Sous-requêtes scalaires, données temporelles.....	59
3.9 Les autres objets SQL.....	73
3.9.1 Les index.....	73
3.9.2 Les vues.....	76
3.9.3 Les fonctions utilisateurs.....	78
3.9.4 Les procédures stockées.....	81
4 Conclusion.....	85
5 Annexe.....	86
5.1 Bibliographie.....	86
5.2 Liens utiles.....	88
5.3 Problème de connexion MariaDB avec Netbeans.....	89
5.4 Problème de configuration avec Phpmyadmin.....	92
6 Changelog.....	94

Notes de l'auteur :

Je m'appelle Grégory Jarrige.

Je suis développeur professionnel depuis 1991. Après avoir longtemps travaillé sur des gros systèmes et des langages et technos propriétaires, j'ai fait le pari de me former aux technos et langage open source vers 2005-2006. J'ai commencé à développer des applications webs professionnelles à partir de 2007, avant d'en faire mon activité principale à partir de 2010. L'arrivée du HTML5 dans la même période a été pour moi une véritable bénédiction, et surtout un formidable terrain d'expérimentation (avec des API comme Canvas, WebAudio, etc...).

En plus de mon activité de développeur freelance, je suis également formateur - sur des sujets tels que PHP, HTML5, Javascript, SQL - tantôt en entreprise, tantôt dans le cadre de programmes de reconversion (GRETA notamment).

Ce support est une création réalisée en décembre 2017 en vue de proposer une introduction rapide à la norme SQL pour des personnes débutant en programmation. L'essentiel des exemples sont axés sur MySQL et sur MariaDB.

Ce document est disponible en téléchargement libre sur mon compte Github :

<https://github.com/gregja/SQLCorner>

Il est publié sous Licence Creative Commons n° 6.

1 Introduction

1.1 Définition

Extraits de la page de présentation Wikipédia :

SQL (sigle de Structured Query Language, en français langage de requête structurée) est un langage informatique normalisé servant à exploiter des bases de données relationnelles. La partie langage de manipulation des données de SQL permet de rechercher, d'ajouter, de modifier ou de supprimer des données dans les bases de données relationnelles.

Créé en 1974, normalisé depuis 1986, le langage est reconnu par la grande majorité des systèmes de gestion de bases de données relationnelles (abrégié SGBDR) du marché.

Source : https://fr.wikipedia.org/wiki/Structured_Query_Language

La norme SQL se compose de plusieurs parties dont les plus importantes sont :

- "DDL" pour "Data Definition Language", qui permet de créer et modifier l'organisation des données dans la base de données (c'est avec des instruction DDL que l'on crée des tables SQL et d'autres types d'objets comme les vues et les procédures stockées).
- "DML" pour "Data Manipulation Language", soit le langage de manipulation des données. C'est avec DML que l'on écrit des requêtes d'interrogation (avec l'ordre « SELECT »), des requêtes de mise à jour (avec l'ordre « UPDATE »), etc...

Ce sont ces deux grandes parties que nous aborderons dans ce tutorial.

Autres parties importantes, mais qui seront peu ou pas abordées dans ce tutorial :

- la partie langage de contrôle de transaction qui permet de commencer et de terminer des transactions,
- la partie langage de contrôle des données qui permet d'autoriser ou d'interdire l'accès à certaines données à certaines personnes.

On notera que la plupart des SGBD implémentent SQL en respectant la norme dans les grande lignes, mais en introduisant des subtilités syntaxiques qui compliquent le portage de code SQL d'un SGBD à l'autre. On trouve des différences syntaxiques à la fois dans DDL et dans DML.

La norme SQL évolue, chaque version porte un numéro correspondant à son année de parution. La dernière version en date est la version SQL:2011.

Pour une présentation plus approfondie et une liste exhaustive des versions de la norme SQL :

https://fr.wikipedia.org/wiki/Structured_Query_Language

1.2 L'outillage

Comme il existe plusieurs implémentations de SQL, il nous faut en choisir une pour débiter. Je vous propose de travailler avec MySQL, et en particulier avec sa déclinaison la plus prometteuse, à savoir MariaDB.

Quelques précisions sur MySQL et MariaDB empruntées à Wikipédia :

MySQL AB a été acheté le 16 janvier 2008 par Sun Microsystems pour un milliard de dollars américains. En 2009, Sun Microsystems a été acquis par Oracle Corporation, mettant entre les mains d'une même société les deux produits concurrents que sont Oracle Database et MySQL. Ce rachat a été autorisé par la Commission européenne le 21 janvier 2010.

Depuis mai 2009, son créateur Michael Widenius a créé un fork de MySQL, qu'il a appelé MariaDB, pour continuer son développement en tant que projet Open Source.

Page de présentation de MariaDB sur Wikipédia :

<https://fr.wikipedia.org/wiki/MariaDB>

Site officiel de MariaDB :

<https://mariadb.com/fr>

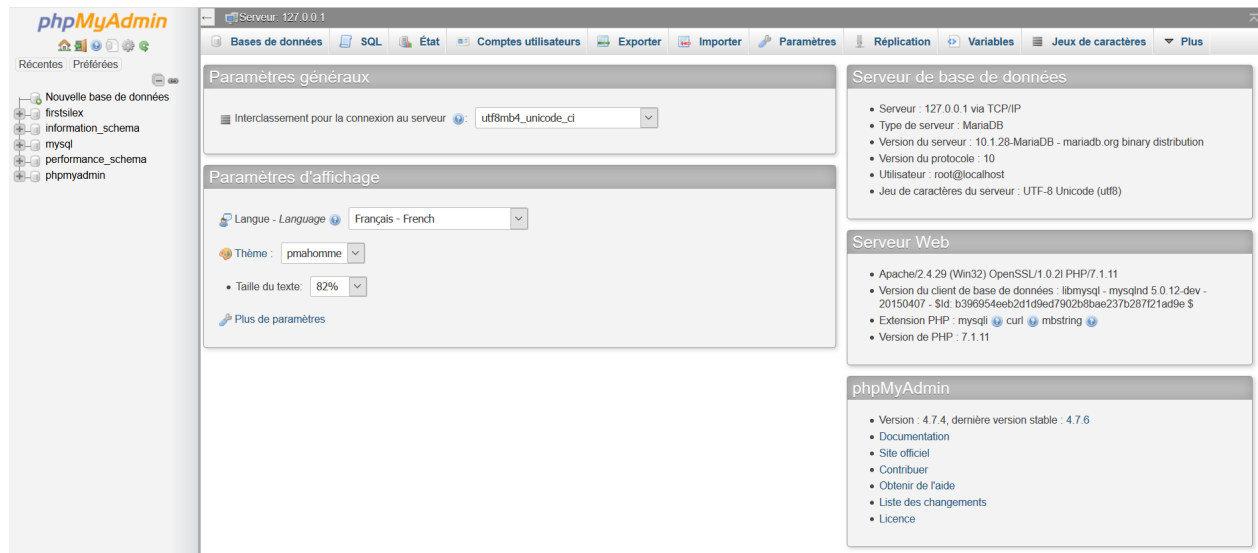
Si vous avez déjà étudié mon cours consacré au langage PHP :

<https://github.com/gregja/PHPCorner>

... alors vous avez très certainement déjà installé un stack PHP comprenant MariaDB.

Si votre stack PHP est un XAMPP en version 7.1.11 (disponible depuis juillet 2017), alors vous disposez de MariaDB en version 10.1.28, ce qui est très satisfaisant pour découvrir de nombreuses possibilités de SQL, dont certaines n'étaient pas disponibles dans les anciennes versions de MySQL.

Dans un stack PHP de base (comme celui fourni par XAMPP ou Wampserver, vous disposez du logiciel PHPMyAdmin. Ce logiciel entre dans la catégorie des « clients SQL ». Il est développé en PHP et c'est un formidable couteau suisse pour travailler avec MySQL et MariaDB. Son interface peut sembler déroutante au début, mais avec un peu de pratique on s'y fait vite :



On peut souligner que PHPMyAdmin est fourni par la plupart des hébergeurs (OVH, Gandi, etc.) pour gérer les bases de données MySQL (ou MariaDB) de leurs clients. C'est donc un outil utilisé par un nombre de personnes... probablement astronomique.

Nous allons voir dans les chapitres suivants comment créer et manipuler une base de données MariaDB.

Site officiel de PHPMyAdmin : <https://www.phpmyadmin.net/>

La dernière version stable de ce soft est la 4.7.6, mais la plupart des stacks PHP ont quelques versions de retard (comme XAMPP qui fournit la 4.7.1). Cela n'a pas d'incidence pour nous, dans le cadre de ce tutorial.

1.3 Pour les lecteurs pressés

Si vous lisez ce document, c'est que vous souhaitez apprendre le SQL, que ce soit par envie ou par nécessité.

J'ai essayé en rédigeant ce document, de laisser de côté le superflu, et de me concentrer sur les éléments qui me semblent essentiels, dans cette norme SQL. Et je l'ai fait avec mon point de vue de développeur d'application « métier », en mettant l'accent sur des cas pratiques, et en éliminant les développements théoriques, que vous pourrez retrouver dans d'autres ouvrages, si vous éprouvez le besoin d'approfondir le sujet.

J'insiste beaucoup sur l'importance de la pratique. Le SQL n'est pas si compliqué que cela, rassurez-vous, mais beaucoup de notions propres au SQL se comprennent très bien avec un petit peu de pratique, alors qu'elles demeurent complètement abstraites si on ne les a pas mises en œuvre soi-même au moins une fois.

SQL peut sembler rébarbatif au premier abord, mais en réalité c'est un outil très puissant et très souple, qui offre un éventail de possibilités énorme, pour qui veut apprendre à les maîtriser.

2 Introduction à SQL

2.1 Création d'une base de test MariaDB

PHPMyAdmin fournit de nombreux assistants qui masquent la complexité de SQL. Ils peuvent sembler pratiques au premier abord, mais en les utilisant on n'apprend pas véritablement à programmer en SQL. On se retrouve dès lors très dépendant de PHPMyAdmin, ce qui va diminuer notre capacité à passer sur d'autres SGBD par la suite. Notre objectif étant d'apprendre à maîtriser SQL, nous allons privilégier une approche DIY (Do It Yourself), donc nous allons passer le plus souvent par l'onglet « SQL » et saisir notre code SQL nous même, comme des grands :



Attention, la table « matable » de l'exemple ci-dessus n'existe pas encore, donc la requête SQL ci-dessus ne pourra fonctionner telle quelle. Nous allons voir dans la suite du tuto comment créer des tables en SQL.

Supposons que nous arrivions pour la toute première fois sur PHPMyAdmin, nous n'avons pas encore créé de base de test nous permettant de travailler. Je vous propose d'en créer une, que nous appellerons « test », tout simplement. Si vous avez déjà une base « test », rien ne vous empêche de créer une base « test2 », à vous de voir.

Voici le code qui va nous permettre de créer une base « test » :

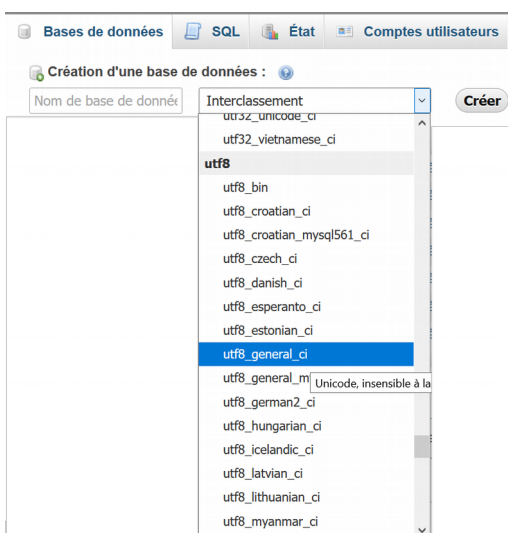
```
CREATE DATABASE test
  CHARACTER SET 'utf8'
  COLLATE 'utf8_general_ci';
```

Le code se compose de 3 lignes qui définissent respectivement :

- le nom de la base (ici « test »)
- l'encodage utilisé par défaut (ici « UTF-8 »)
- le type de tri utilisé, défini par le paramètre « collate » et qui est fixé ici à un type générique associé à l'encodage « UTF-8 ».

En définissant notre base de données de cette manière, toutes les tables que nous créerons par la suite dans cette base hériteront des caractéristiques de la base, sauf si nous les paramétrons avec un encodage différent, de manière explicite. Les paramètres que nous avons définis ci-dessus conviendront dans la plupart des cas, nous aurons rarement besoin de les modifier (sauf si nous travaillons sur des langues très spécifiques nécessitant un encodage particulier).

Si vous êtes amené à travailler avec un encodage particulier, et que vous n'êtes pas sûr de l'encodage à utiliser, le plus simple est de prendre l'assistant de création de base de PHPMyAdmin et de regarder la liste des encodages proposés. Vous verrez que le choix est impressionnant :

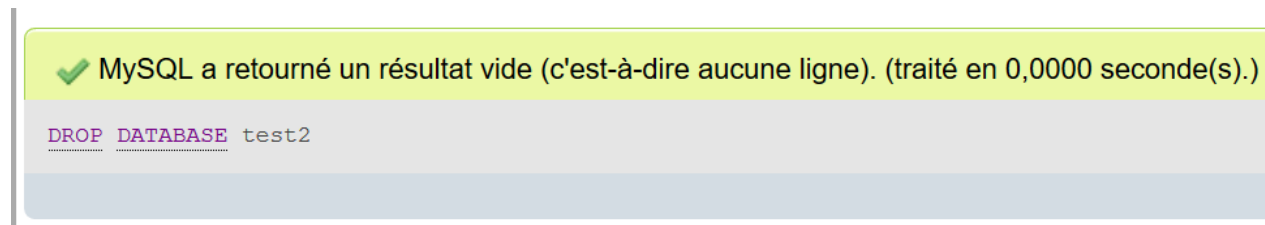


Si vous vous êtes trompé dans le nom de la base de données, et que vous souhaitez la recréer, c'est très simple, saisissez dans la fenêtre SQL l'instruction suivante :

```
DROP DATABASE test2
```

et cliquez sur le bouton « Exécuter » (ou « Go » si vous utilisez une version non francisée) qui se situe en bas à droite de la fenêtre.

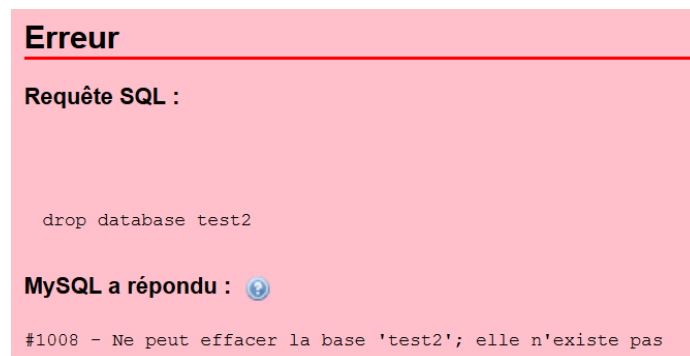
Vous verrez apparaître un message de ce type (ou un équivalent en anglais) :



On notera que PHPMyAdmin ne sait pas distinguer MySQL de MariaDB, donc il vous dira toujours que « MySQL a fait quelque chose... », même si c'est MariaDB qui « tourne » dans les coulisses.

Donc PHPMyAdmin nous indique que « MySQL a retourné un résultat vide ». C'est normal car cette requête est destinée à supprimer un objet SQL, elle n'est pas sensée renvoyer un résultat quel qu'il soit.

Si vous vous étiez trompé, en utilisant un nom de base de données inexistant, vous auriez obtenu un message très différent :



A noter : dans la suite du tuto, j'utiliserai souvent les majuscules pour les ordres SQL, et les minuscules pour les noms d'objets comme dans l'exemple suivant :

```
DROP DATABASE test3
```

C'est une convention que j'ai adoptée arbitrairement, pour vous aider à distinguer les ordres SQL des noms d'objets. En réalité, SQL est très permissif au niveau de la syntaxe et vous n'avez pas obligation de suivre cette convention. Vous pouvez écrire l'instruction ci-dessus intégralement en minuscules si vous le souhaitez.

2.2 Création d'une table pivot

Dans quelques cas particuliers, nous serons amenés à utiliser une table pivot.

Il s'agit d'une table contenant une seule ligne et une seule colonne, elle aura un usage purement technique que nous étudierons plus tard. Pour l'heure, cette table pivot va nous permettre de découvrir comment on crée une table en SQL.

NB : pour la lisibilité du code SQL, nous placerons souvent des commentaires. Les commentaires en SQL se définissent en plaçant 2 tirets en début de ligne. Nous allons en voir un exemple tout de suite avec la création de la table « pivot » :

```
-- Structure de la table pivot
CREATE TABLE pivot (
  colx INT(1) UNSIGNED NOT NULL
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

Nous avons ici un exemple typique de création d'objet SQL de type TABLE. Il existe d'autres types d'objets en SQL, que nous étudierons plus tard.

La table créée ici s'appelle « pivot », elle contient une seule colonne que nous avons appelée « colx ». Cette colonne est de type « integer », mais un type « integer » particulier puisqu'il ne peut contenir qu'un seul chiffre non signé, et qu'il ne peut pas contenir de valeur nulle.

La table créée ici utilise un moteur SQL qui s'appelle InnoDB et elle va utiliser comme encodage l'UTF-8 par défaut. Si on avait oublié de préciser l'encodage, c'est celui défini au niveau de la base qui aurait été utilisé, donc dans notre cas cela aurait été le même encodage.

C'est quoi ce moteur InnoDB. Il faut savoir que MariaDB fournit plusieurs moteurs de stockage, qui répondent à des besoins différents. C'est une particularité de MySQL - et de MariaDB - de proposer plusieurs moteurs de stockage, les autres SGBD du marché ne proposant pas ce type d'approche. Les moteurs les plus couramment utilisés avec MySQL et MariaDB sont InnoDB et MyISAM. Dans le cadre de ce tuto, vous pourrez utiliser indifféremment InnoDB ou MyISAM car nous n'aborderons pas, dans cette version du tuto, l'étude des avantages de InnoDB (ce sera peut être pour une version ultérieure).

On trouve de nombreux articles expliquant les différences entre les moteurs MariaDB, je vous laisse le soin de les étudier :

<http://sql.sh/1548-mysql-innodb-mysam>

<http://www.tux-planet.fr/mysql-les-principales-differences-entre-mysam-et-innodb/>

Nous avons créé notre table pivot, nous devons maintenant y injecter de la donnée. Nous allons y injecter une ligne, avec l'instruction SQL INSERT :

```
-- Injection d'une ligne dans la table pivot
INSERT INTO pivot (colx)
VALUES
(1);
```

Nous avons indiqué à SQL que nous injectons une ligne contenant une seule colonne (colx), et que nous voulons injecter dans cette colonne la valeur 1 (définie sur la dernière ligne).

Ne vous y trompez pas, la valeur « 1 » définit la valeur injectée dans la colonne « colx », il ne s'agit pas d'un numéro de ligne. Si nous avions voulu injecter 2 lignes, avec dans la colonne « colx » :

- la valeur 1 pour la première ligne,
- et la valeur 3 pour la seconde ligne

... nous aurions écrit ceci :

```
INSERT INTO pivot (colx)
VALUES
(1),
(3);
```

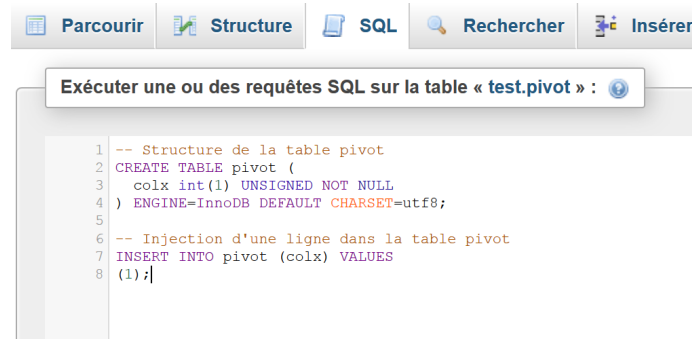
Les lignes injectées dans la table « pivot » sont définies après le mot réservé « VALUES ».

On voit que chaque ligne injectée dans la table est encadrée par un jeu de parenthèses, et que ces jeux de parenthèses sont séparés par des virgules.

La syntaxe de SQL est très rigide, ne vous trompez pas dans l'emplacement des virgules et des parenthèses. En revanche SQL est très souple en ce qui concerne l'emplacement des sauts de ligne. Donc l'instruction ci-dessous est strictement équivalente à l'instruction précédente :

```
INSERT INTO pivot (colx) VALUES (1),(3);
```

A noter que le point virgule en fin de ligne est facultatif. Il n'est utile que dans le cas où vous souhaitez exécuter plusieurs instructions en une seule passe, comme dans l'exemple suivant:



Si l'on part du principe que l'injection de donnée que j'ai effectuée dans ma table est la suivante

```
INSERT INTO pivot (colx) VALUES (1);
```

Jetons un coup d'oeil au contenu de notre table « pivot » avec l'instruction SQL suivante :

```
SELECT * FROM pivot
```

Vous allez voir apparaître le résultat suivant dans PHPMyAdmin :

colx
1

Notre table contient une seule colonne « colx » et une seule ligne. Dans cette unique ligne, la colonne « colx » contient la valeur numérique 1. Voilà, c'est aussi simple que ça.

Nous venons de voir que l'instruction SELECT nous permet de consulter le contenu d'une table. En écrivant ceci :

```
SELECT * FROM pivot
```

... nous indiquons à l'interpréteur SQL de MariaDB que nous souhaitons afficher toutes les colonnes (c'est l'astérisque qui permet de le préciser) de la table « pivot ». Notre table « pivot » n'a pas un contenu suffisamment intéressant pour que nous approfondissions l'instruction « SELECT », nous le ferons donc plus tard.

Regardons plutôt comment faire pour supprimer notre table « pivot », si besoin :

```
DROP TABLE pivot
```

Exécutez cette instruction via PHPMyAdmin... et voilà, plus de table « pivot ».

Avant de passer à la suite, nous allons recréer cette table pivot, mais je vais en profiter pour vous montrer une astuce. Pour créer la table « pivot », nous avons d'abord exécuté une instruction CREATE TABLE, puis une instruction INSERT. Eh bien nous pouvons faire du « 2 en 1 » avec l'instruction suivante, que je vous invite à tester :

```
CREATE TABLE pivot  
SELECT CAST(1 as unsigned integer) as colx
```

Dans l'instruction ci-dessus, la table « pivot » est créée « à la volée » à partir du jeu de données renvoyé par la requête « SELECT » qui se trouve juste derrière (vous noterez qu'il n'y a pas de point virgule entre les 2 lignes de code, du coup l'interpréteur SQL considère qu'elles constituent une seule et même instruction SQL.

La particularité de cette technique, c'est qu'elle ne permet pas de préciser l'encodage et le moteur (InnoDB ou autre) utilisé. Ce n'est pas gênant dans le cas présent, aussi nous allons en rester là pour l'instant.

Notez que nous pourrions réutiliser cette technique consistant à générer une table à partir du résultat d'une requête. Elle rend quelquefois de précieux services.

Nous avons vu ici comment créer une table très simple : une seule ligne et une seule colonne, c'est un cas très particulier. Nous verrons dans les chapitres suivants des exemples de tables plus riches en termes de structure et de contenu.

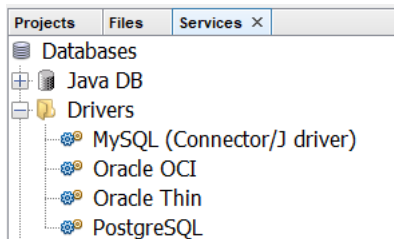
2.3 Quelques outils SQL à connaître

Avant de poursuivre notre exploration de SQL, parlons brièvement des outils qui sont à notre disposition. Car PHPMyAdmin n'est pas le seul outil que nous pouvons utiliser pour gérer nos bases MariaDB.

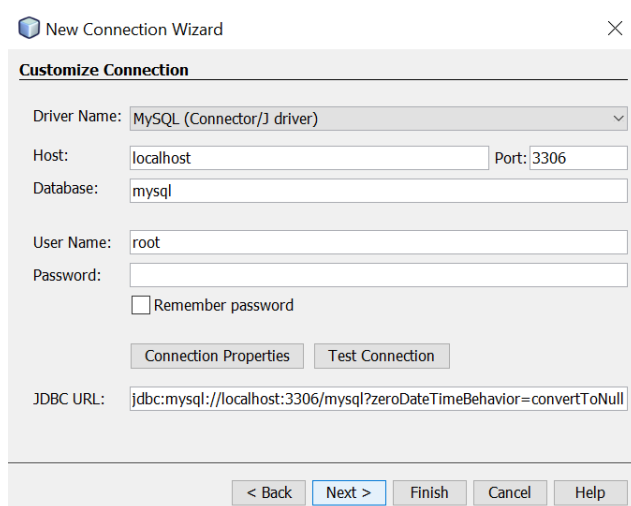
Attention : pour la plupart des outils cités dans ce chapitre, je recommande de toujours télécharger les outils en passant par le site officiel du projet concerné. Il faut se méfier des sites non officiels qui proposent des packages de projets dont ils ne sont pas maîtres d'oeuvre. On risque au mieux de récupérer un produit obsolète, au pire de récupérer un produit infecté par un virus.

2.3.1 Netbeans

L'IDE d'Oracle fournit un client SQL de bonne facture, qui est capable de travailler avec plusieurs SGBD (MySQL et MariaDB, PostgreSQL, DB2, Oracle). On y accède via l'onglet « Services » option « Databases ».



La connexion à MySQL se fait simplement : un clic droit sur « MySQL » puis option « Connect Using » et on arrive sur l'écran de définition de la connexion :



Modifiez « user name » et « password » si nécessaire, cliquez sur « Finish ».

Faites ensuite un clic-droit sur la ligne « jdbc:mysql... », option « Execute Command », ce qui va avoir pour effet d'ouvrir un éditeur SQL dans un nouvel onglet, comme celui qui s'appelle « SQL1 » dans l'exemple suivant :

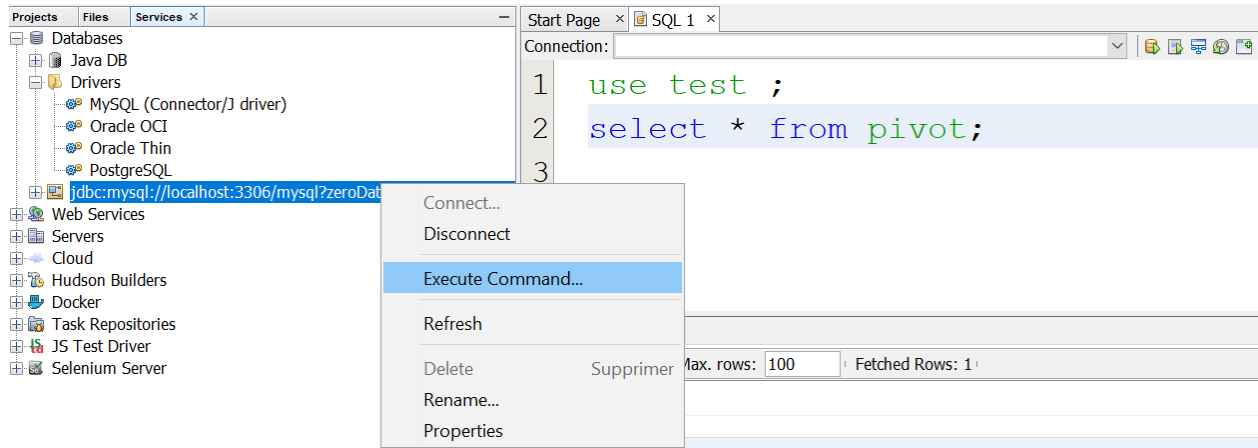
Attention : si en cliquant sur le bouton « Test Connection », vous voyez apparaître le message d'erreur suivant :

*Cannot establish a connection to jdbc:mysql://localhost:3306/mysql?
zeroDateTimeBehavior=convertToNull using com.mysql.jdbc.Driver*

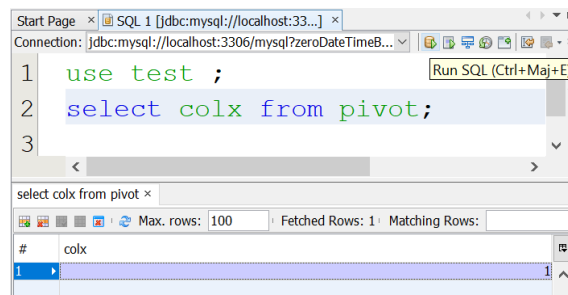
Alors vous avez un problème de configuration de firewall. C'est un problème que vous êtes susceptible de rencontrer sur certains systèmes sous Linux, dont le Mac. Prière de vous reporter

à l'annexe (5.3) pour de plus amples précisions sur la manière de corriger le problème.

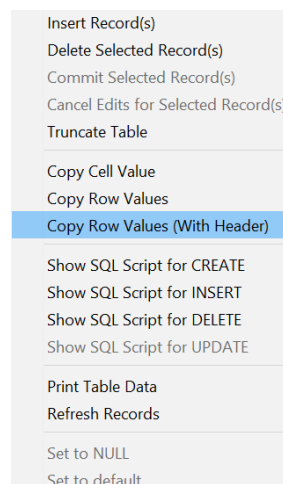
Si votre configuration est correcte et que votre connexion avec MariaDB se fait bien, alors vous pouvez commencer à exécuter des instructions SQL, comme dans l'exemple suivant :



Dans les exemples ci-dessus et ci-dessous, l'instruction « use test » permet d'indiquer à Netbeans que l'on se connecte sur la base « test ». A partir de là, le SELECT qui suit va pouvoir fonctionner. Cliquez sur l'icône « Run SQL », le résultat apparaît en dessous :



Un clic-droit sur le jeu de données obtenu permet d'accéder à des options intéressantes :



... comme par exemple l'option « Copy Row Values (With Header) » qui permet de récupérer le jeu de données pour le coller facilement dans un tableau.

2.3.2 Clients SQL alternatifs

Parmi les clients SQL alternatifs, on trouve :

- SquirrelSQL : <http://squirrel-sql.sourceforge.net/>
- Dbeaver : <https://github.com/serge-rider/dbeaver/wiki>

Ces deux outils ont l'avantage d'être extensibles via des plugins, de supporter de nombreux SGBD, et d'être multiplateformes.

SquirrelSQL est un bon outil que j'ai déjà eu l'occasion d'utiliser. Je n'ai pas testé Dbeaver qui me semble très intéressant également.

Je sais que SquirrelSQL est apprécié par de nombreux administrateurs de bases de données, du fait qu'il leur permet de disposer d'un seul outil pour administrer des bases fonctionnant sur plusieurs SGBD (ce qui arrive fréquemment en entreprise).

2.3.3 Outils de modélisation

Les solutions permettant de modéliser une base de données, soit au niveau d'un modèle conceptuel de données (MCD), soit au niveau d'un modèle physique de données (MPD), ne sont pas très nombreuses sur le marché. On trouve surtout des solutions propriétaires onéreuses comme PowerDesigner (anciennement PowerAMC), commercialisé par SAP :

<https://fr.wikipedia.org/wiki/PowerAMC>

J'avais eu l'occasion de travailler avec PowerAMC au début des années 2000, c'était un produit puissant, répondant bien à certains besoins, mais avec un tarif « puissant » lui aussi.

Apparu plus récemment, Vertabelo se présente comme une solution de modélisation de base de données permettant le travail en équipe, avec une version « academic » (gratuite pour les étudiants et les enseignants) :

<https://www.vertabelo.com/>

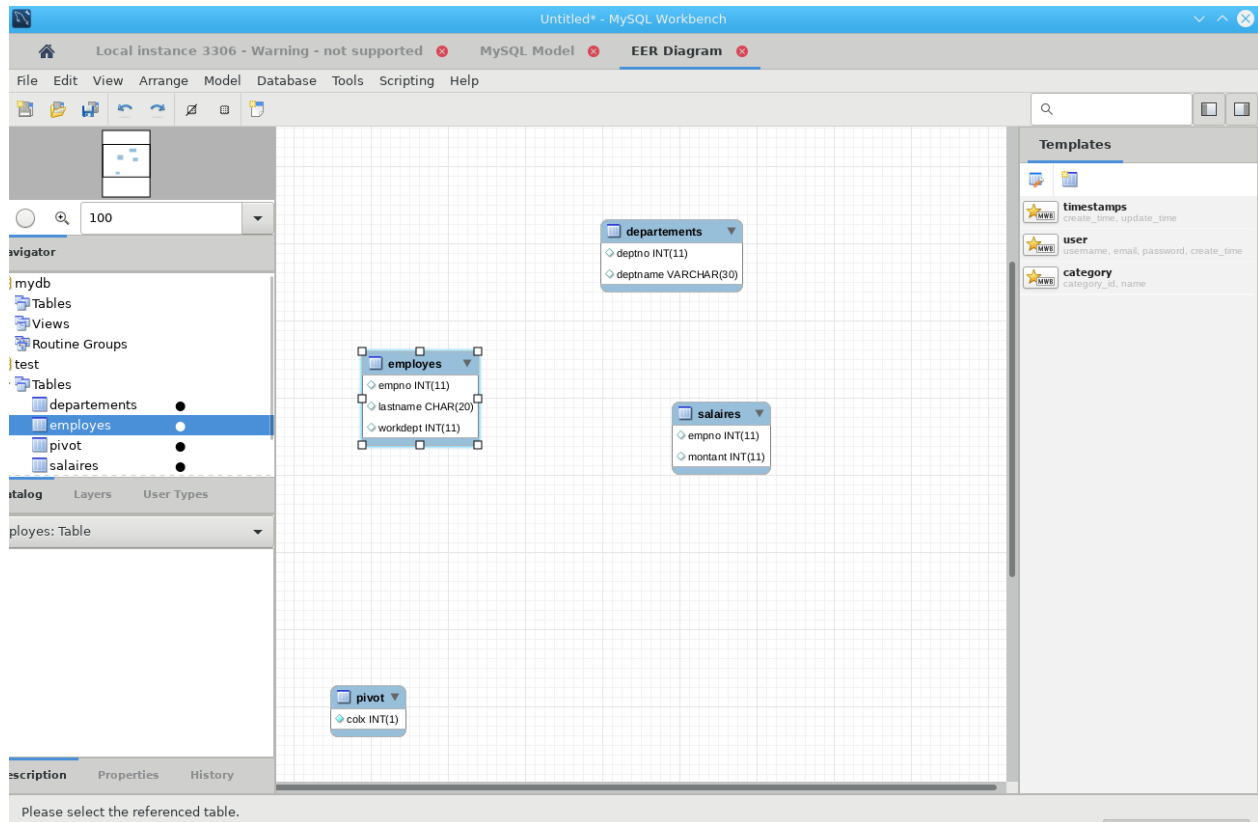
C'est une solution qui semble intéressante, mais je ne l'ai pas testée (peut être dans un avenir proche).

Dans les solutions de modélisation open source, il n'y a guère que MySQL Workbench qui tienne encore la route (la plupart des projets concurrents ayant disparu). C'est un outil graphique, capable d'importer une base MySQL existante, de la restructurer (par exemple en redéfinissant

des dépendances entre des tables), avant de générer le code SQL d'altération des tables modifiées. Cette solution très appréciée des développeurs MySQL est encore disponible en téléchargement, mais elle n'est plus réellement maintenue, et son installation sur des postes sous Windows (récent) est devenue problématique, voire impossible. L'installation sous Linux est encore possible :

<https://www.mysql.com/fr/products/workbench/>

Testé sous Linux Mageia, voici l'affichage du diagramme EER (*) :



(*) The **enhanced entity-relationship (EER)** model (or **extended entity-relationship** model) in computer science is a high-level or conceptual data model incorporating extensions to the original entity-relationship (ER) model, used in the design of databases.

(source : https://en.wikipedia.org/wiki/Enhanced_entity%E2%80%93relationship_model)

Pour une présentation en français du même sujet :

https://fr.wikipedia.org/wiki/Mod%C3%A8le_entit%C3%A9-association

2.3.4 Commandes MariaDB utiles

Connaître la version de MariaDB (ou MySQL) utilisée :

```
SHOW VARIABLES LIKE "%version%";
```

Variable name	Value
innodb_version	5.6.36-82.2
protocol_version	10
slave_type_conversions	
version	10.1.28-MariaDB
version_comment	mariadb.org binary distribution
version_compile_machine	32
version_compile_os	Win32
version_malloc_library	system
version_ssl_library	YaSSL 2.4.2

Connaître la liste des bases de données MariaDB disponible :

```
SHOW DATABASES;
```

Database
information_schema
mysql
performance_schema
phpmyadmin
test

Connaître la liste des tables à l'intérieur d'une base (par exemple la base « test ») :

```
SHOW TABLES FROM test;
```

Tables_in_test
employes
pivot

Connaître la structure d'une table :

```
SHOW columns FROM employes;
```

Field	Type	Null	Key	Default	Extra
EMPNO	char(6)	YES			
LASTNAME	char(20)	YES			
WORKDEPT	char(6)	YES			

Variante de SHOW COLUMNS qui fournit le même niveau de détail :

```
DESCRIBE employes;
```

Variante de SHOW COLUMNS et DESCRIBE, qui permet de personnaliser le format des données récupérées :

```
SELECT COLUMN_NAME, ORDINAL_POSITION as POS,  
       COLUMN_TYPE, CHARACTER_SET_NAME, COLLATION_NAME, COLUMN_COMMENT  
FROM INFORMATION_SCHEMA.COLUMNS  
WHERE table_name = 'employes'  
AND table_schema = 'test'
```

COLUMN_NAME	POS	COLUMN_TYPE	CHARACTER_SET_NAME	COLLATION_NAME	COLUMN_COMMENT
EMPNO	1	char(6)	utf8	utf8_general_ci	
LASTNAME	2	char(20)	utf8	utf8_general_ci	
WORKDEPT	3	char(6)	utf8	utf8_general_ci	

Consulter la liste des index d'une table :

```
SHOW index FROM employes
```

2.3.5 Outil pour la création de jeux de données

Le site internet Mockaroo peut constituer un précieux allié pour constituer des jeux de données cohérents, en attendant de disposer des jeux de données réels nécessaires à un projet en cours de développement.

Site : <http://mockaroo.com/>

On peut personnaliser la structure du jeu de données généré, comme dans l'exemple ci-dessous ou j'ai remplacé la colonne « adresse IP » par la colonne « départements ». On peut demander à obtenir le jeu de données sous forme de code SQL (ou CSV, JSON, etc..), et on peut même obtenir le code SQL de création de la table en cochant l'option « include create table » :

The screenshot shows the Mockaroo 'realistic data generator' interface. It features a table with columns: Field Name, Type, and Options. The fields are: id (Row Number), first_name (First Name), last_name (Last Name), email (Email Address), gender (Gender), and department (Department (Corp ...)). Each field has a 'blank' percentage set to 0 and a 'fx' icon. Below the table is an 'Add another field' button. At the bottom, there are settings for '# Rows' (10), 'Format' (SQL), 'Table Name' (employees), and a checked 'include create table' checkbox. A green 'Download Data' button is on the left, and a 'Preview' button is next to it. To the right, there is a 'More' dropdown and a link to 'Sign up for free'.

Field Name	Type	Options
id	Row Number	blank: 0 % fx ✕
first_name	First Name	blank: 0 % fx ✕
last_name	Last Name	blank: 0 % fx ✕
email	Email Address	blank: 0 % fx ✕
gender	Gender	blank: 0 % fx ✕
department	Department (Corp ...)	blank: 0 % fx ✕

Add another field

Rows: 10 Format: SQL Table Name: employees ☒ include create table

Download Data Preview More Want to save this for later? [Sign up for free.](#)

Exemple de code généré par Mockaroo :

```
create table employes (
  id INT,
  first_name VARCHAR(50),
  last_name VARCHAR(50),
  email VARCHAR(50),
  gender VARCHAR(50),
  departements VARCHAR(50)
);
```

```
insert into employees (id, first_name, last_name, email, gender, departements) values
(1, 'Salomo', 'Shardlow', 'sshardlow0@sakura.ne.jp', 'Male', 'Services');

insert into employees (id, first_name, last_name, email, gender, departements) values
(2, 'Danice', 'Leadbeater', 'dleadbeater1@amazon.com', 'Female', 'Accounting');

insert into employees (id, first_name, last_name, email, gender, departements) values
(3, 'Cointon', 'MacIntosh', 'cmacintosh2@si.edu', 'Male', 'Human Resources');

insert into employees (id, first_name, last_name, email, gender, departements) values
(4, 'Hannie', 'Coulthard', 'hcoulthard3@mtv.com', 'Female', 'Business Development');

insert into employees (id, first_name, last_name, email, gender, departements) values
(5, 'Craggie', 'Habgood', 'chabgood4@github.io', 'Male', 'Accounting');

insert into employees (id, first_name, last_name, email, gender, departements) values
(6, 'Bobbi', 'Temprell', 'btemprell5@google.fr', 'Female', 'Sales');

insert into employees (id, first_name, last_name, email, gender, departements) values
(7, 'Joachim', 'Donovan', 'jdonovan6@businesswire.com', 'Male', 'Human Resources');

insert into employees (id, first_name, last_name, email, gender, departements) values
(8, 'Orion', 'Schwander', 'oschwander7@phoca.cz', 'Male', 'Research and Development');

insert into employees (id, first_name, last_name, email, gender, departements) values
(9, 'Kaila', 'Phethean', 'kphethean8@cdc.gov', 'Female', 'Legal');

insert into employees (id, first_name, last_name, email, gender, departements) values
(10, 'Gan', 'Newnham', 'gnewnham9@wunderground.com', 'Male', 'Legal');
```


3 Les bases de SQL

3.1 Créer un jeu de données métier

Pour cette première phase de découverte, nous allons générer un jeu de données simplifié, avec une table des employés, une table des départements, une table des salaires, etc..

Exemple de table SQL contenant 3 colonnes (« numéro d'employé », « nom de famille », « numéro de département » dans lequel la personne travaille) :

```
CREATE TABLE employes (  
    empno INTEGER,  
    lastname CHAR(20),  
    workdept INTEGER  
);
```

Sur le même principe que la table pivot vue au début de ce tuto, nous injectons ici un jeu de quelques lignes, chaque ligne contenant les 3 colonnes définies lors du CREATE TABLE.

```
INSERT INTO employes (empno, lastname, workdept)  
VALUES  
(20, 'THOMPSON', 1),  
(60, 'STERN', 2),  
(100, 'SPENSER', 3),  
(170, 'YOSHIMURA', 2),  
(180, 'SCOUTTEN', 2),  
(190, 'WALKER', 2),  
(250, 'SMITH', 2),  
(280, 'SCHNEIDER', 5),  
(300, 'SMITH', 5),  
(310, 'SETRIGHT', 5)  
;
```

Une table contenant les salaires des employés :

```
CREATE TABLE salaires (  
    empno INTEGER,  
    montant INTEGER  
);
```

```
INSERT INTO salaires (empno, montant)  
VALUES  
(20, 45000),  
(60, 42000),  
(100, 41000),  
(170, 35000),
```

```
(180, 27500),  
(190, 56823),  
(250, 75248),  
(280, 36150),  
(300, 41563),  
(310, 51234)  
;
```

Et pour finir, une table définissant les départements dans lesquels les employés travaillent :

```
CREATE TABLE departements (  
    deptno INTEGER,  
    deptname VARCHAR(30)  
);  
  
INSERT INTO departements  
(deptno, deptname)  
VALUES  
(1, 'PLANNING'),  
(2, 'MANUFACTURING SYSTEMS'),  
(3, 'SOFTWARE SUPPORT'),  
(4, 'ADMINISTRATION SYSTEMS'),  
(5, 'OPERATIONS');
```

Petite anecdote : en créant la première version de la table « departements », je m'étais trompé, en mettant un nom à l'anglaise (« department » sans accent), ce qui n'était pas très cohérent par rapport à ma table des salaires et à ma table des employés. Pour rectifier le tir, on peut bien évidemment modifier le code de création de table ci-dessus, mais on peut aussi dupliquer la table sous un nouveau nom, en utilisant la technique suivante :

```
CREATE TABLE departements  
SELECT * FROM department ;
```

Ne pas oublier de supprimer l'ancienne table par un DROP une fois la duplication terminée :

```
DROP TABLE department ;
```

Bon, il est vrai que ce genre de correction est plus facile à appliquer quand une table est encore peu utilisée. De plus, ma table « department » n'était dépendante d'aucun autre objet SQL (index ou vue), donc le remplacement de table est facile à faire dans ce cas. Nous évoquerons plus tard les notions d'indexs et de vues, pas d'inquiétude.

3.2 Requêtes de type *SELECT* simples

Commençons par une requête simple sur la tables des employés :

```
SELECT * FROM employes
```

EMPNO	LASTNAME	WORKDEPT
20	THOMPSON	1
60	STERN	2
100	SPENSER	3
170	YOSHIMURA	2
180	SCOUTTEN	2
190	WALKER	2
250	SMITH	2
280	SCHNEIDER	5
300	SMITH	5
310	SETRIGHT	5

Nous pouvons filtrer avec la clause WHERE, pour n'afficher que...

- ... les employés du département n° 1 :

```
SELECT * FROM employes WHERE workdept = 1
```

- ... les employés des départements n° 1 ou 3 :

```
SELECT * FROM employes WHERE workdept = 1 OR workdept = 3
```

EMPNO	LASTNAME	WORKDEPT
20	THOMPSON	1
100	SPENSER	3

- ... les employés des départements n° 1 et 3 :

```
SELECT * FROM employes WHERE workdept = 1 AND workdept = 3
```

Ah, tiens, cette requête ne renvoie rien ?!? Eh oui, c'est normal, car aucun employé n'est sur 2 départements en même temps. Or dans notre condition, nous demandons à afficher les employés qui sont affectés au département 1 et au département 3. C'est une condition impossible à réaliser compte tenu de la structure de notre table « employes ». L'interpréteur SQL ne renverra pas d'erreur dans un cas comme celui-là, il nous renverra simplement un jeu de données vide. C'est à nous de faire attention et de définir une requête qui soit en phase avec la structure de nos tables.

Tiens, une petite astuce à noter : plutôt que d'écrire le code suivant

```
SELECT * FROM employes WHERE workdept = 1 OR workdept = 3
```

vous pouvez écrire ceci, qui est strictement équivalent :

```
SELECT * FROM employes WHERE workdept IN (2, 3)
```

Le « IN » vous évite de devoir répéter plusieurs fois la colonne « workdept » dans la clause WHERE. Il vous permet de définir une liste des valeurs recherchées, liste que vous lui transmettez entre parenthèses. Vous obtenez ainsi un code plus concis, pour un résultat identique.

Autre technique de filtrage intéressante, la clause LIKE qui permet d'afficher les employés...

- dont le nom commence par « THOM » :

```
SELECT * FROM employes WHERE lastname LIKE 'THOM%'
```

- dont le nom contient « HOM » :

```
SELECT * FROM employes WHERE lastname LIKE '%HOM%'
```

- dont le nom se termine par « SON » :

```
SELECT * FROM employes WHERE lastname LIKE '%SON'
```

On n'est pas obligé de passer par LIKE, si on connaît précisément la position d'une chaîne de caractères, on peut écrire ceci :

```
SELECT * FROM employes WHERE SUBSTR(lastname, 1, 4) = 'THOM'
```

La fonction « SUBSTR » fait partie de la liste des fonctions de manipulation de chaînes de caractères fournie par MariaDB, nous étudierons la liste des fonctions dans un prochain chapitre.

Nous pouvons ajouter la clause ORDER BY, pour trier les employés par département et nom :

```
SELECT * FROM employes  
ORDER BY workdept, lastname
```

EMPNO	LASTNAME	WORKDEPT
20	THOMPSON	1
180	SCOUTTEN	2
250	SMITH	2
60	STERN	2
190	WALKER	2
170	YOSHIMURA	2
100	SPENSER	3
280	SCHNEIDER	5
310	SETRIGHT	5
300	SMITH	5

Par défaut la clause ORDER BY effectue un tri ascendant des données.

```
SELECT * FROM employes
ORDER BY workdept ASC, lastname ASC
```

On pourrait demander un tri des départements ascendant, couplé avec un tri des noms descendant, cela donnerait ceci (que je vous invite à tester) :

```
SELECT * FROM employes
ORDER BY workdept ASC, lastname DESC
```

Nous pouvons demander un comptage du nombre d'employés par département, grâce à la combinaison de :

- la clause GROUP BY, qui nous permet de grouper les employés par département,
- la fonction count(*) qui nous permet d'effectuer un comptage du nombre d'éléments pour chaque département (vous noterez la clause « AS comptage » qui nous permet d'attribuer à la colonne résultat un nom de notre choix)

```
SELECT workdept, count(*) as comptage
FROM employes
GROUP BY workdept
```

workdept	comptage
1	1
2	5
3	1
5	3

Vous noterez la présence de la clause « AS comptage » qui consiste à renommer la colonne résultant de la fonction « count(*) ». Je vous invite à tester la requête ci-dessus sans la clause « AS comptage », vous verrez la différence. Je vous encourage à toujours renommer les colonnes résultant du traitement d'une fonction, plutôt que de laisser SQL attribuer le nom qui l'arrange. Votre code gagnera en clarté, et sera plus facile à intégrer dans une application PHP par

exemple... sachant qu'en PHP, on récupère les données sous forme de tableau associatif dont chaque poste correspond au nom de la colonne renvoyé par SQL.

Bon, elle n'est pas pas mal cette requête de comptage des employés par département, mais ce serait plus sympa si nous affichions, en face de chaque numéro de département, son libellé.

Il y a plusieurs manières de faire cela, voici une première technique qui consiste à ajouter une petite sous-requête à l'intérieur de la requête principale. Il s'agit d'une sous-requête scalaire, c'est à dire une sous-requête qui ne renvoie qu'une ligne et une seule colonne. Dans notre cas, c'est la partie en jaune dans la requête suivante :

```
SELECT workdept,  
      (SELECT deptname FROM departements WHERE deptno = workdept) as  
      libelle_dept,  
count(*) as comptage  
FROM employes  
GROUP BY workdept
```

workdept	libelle_dept	comptage
1	PLANNING	1
2	MANUFACTURING SYSTEMS	5
3	SOFTWARE SUPPORT	1
5	OPERATIONS	3

Vous remarquerez que j'ai pris soin de donner un nom à la colonne générée par la sous-requête scalaire. Je l'ai appelée « libelle_dept », via la clause « as libelle_dept ».

Mais la partie la plus importante de la sous-requête scalaire, c'est la clause WHERE qui va nous permettre de lier, par une condition d'égalité, la colonne « deptno » de la table « departements » avec la colonne « workdept » de la table « employes ».

Prenez le temps d'étudier cette technique, qui est très pratique. Je vous invite à introduire volontairement des erreurs pour voir ce qui se passe. Par exemple, modifiez la sous-requête en ajoutant la colonne « deptno » de la façon suivante :

```
SELECT workdept,  
      (SELECT deptno, deptname FROM departements WHERE deptno =  
workdept) as libelle_dept,  
count(*) as comptage  
FROM employes  
GROUP BY workdept
```

Vous allez voir que SQL va vous renvoyer l'erreur suivante :

#1241 - Operand should contain 1 column(s)

... ce qui confirme bien qu'une sous-requête scalaire ne peut renvoyer qu'une seule colonne.

Autre erreur possible, la sous-requête qui renvoie plusieurs lignes. On peut dans notre cas provoquer cette erreur artificiellement en remplaçant le « égal » du WHERE par un « différent » :

```
SELECT workdept,  
       (SELECT deptname FROM departements WHERE deptno <> workdept) as  
libelle_dept,  
count(*) as comptage  
FROM employes  
GROUP BY workdept
```

Vous allez obtenir l'erreur SQL suivante :

#1242 - Subquery returns more than 1 row

... ce qui confirme bien qu'une sous-requête scalaire ne peut renvoyer qu'une seule ligne.

Vous serez quelquefois confronté à des cas où votre sous-requête renvoie forcément plusieurs lignes. Cela peut arriver, en fonction de la nature de votre jeu de données. Vous pourrez contourner le problème en ajoutant la clause « LIMIT 1 », qui oblige SQL à ne renvoyer que la première ligne trouvée.

Dans le cas de notre requête (dans sa version non boguée), la clause « LIMIT 1 » est superflue, mais voici néanmoins le principe :

```
SELECT workdept,  
       (SELECT deptname FROM departements WHERE deptno = workdept  
        LIMIT 1) as libelle_dept,  
count(*) as comptage  
FROM employes  
GROUP BY workdept
```

Je vous ai montré une technique de comptage consistant à utiliser comme table principale la table des employés, et allant récupérer le libellé des départements dans un second temps via une sous-requête.

On peut utiliser une autre approche consistant à partir de la table des départements (comme table principale), et à effectuer le comptage des employés dans la sous-requête. On inverse en somme le processus, ce qui nous donne ceci :

```
SELECT deptno, deptname,  
       (SELECT count(*) FROM employes WHERE deptno = workdept) as comptage  
FROM departements  
ORDER BY deptno
```

Le résultat obtenu est le suivant :

deptno	deptname	comptage
1	PLANNING	1
2	MANUFACTURING SYSTEMS	5
3	SOFTWARE SUPPORT	1
4	ADMINISTRATION SYSTEMS	0
5	OPERATIONS	3

Ce qui est intéressant à souligner ici, c'est la présence du département n° 4, qui a zéro employé. On ne le voyait pas apparaître dans la requête précédente, et pour cause, la table des employés était la table de référence et aucun employé n'était affecté à ce département n° 4.

Dans quel cas faut-il utiliser la technique que nous venons de voir, dans quel cas faut-il utiliser la technique précédente ? Et bien cela dépend... du résultat attendu, et aussi cela dépend de la volumétrie.

Imaginons en effet que notre table des départements contienne beaucoup de départements, ce qui pourrait être le cas dans une très grosse société. Supposons également que nous ayons beaucoup de départements qui ont été fermés, et donc sur lesquels il n'y a plus d'employés affectés. Si nous souhaitons ne voir apparaître dans notre résultat que les départements ayant des employés, c'est dommage de partir de la table des départements, comme table principale, car on va lire beaucoup de lignes « départements » inutilement. Il sera dans ce cas plus judicieux d'utiliser la table « employes » comme table principale, et donc d'utiliser la première technique. Si au contraire nous voulons voir apparaître tous les départements, y compris ceux qui n'ont plus d'employés, alors la dernière technique que nous avons vue est plus pertinente. Vous voyez, cela dépend... du contexte.

Ce qui est certain en revanche, c'est qu'un bon développeur SQL s'efforcera de rechercher la technique permettant d'obtenir le résultat final souhaité, en minimisant le nombre de lectures dans les tables. La lecture de données inutiles est énergivore, et peut occasionner des lenteurs dans la restitution des résultats. Le développeur SQL recherche donc le meilleur compromis, cela nécessite une bonne connaissance de SQL, et de l'expérience.

3.3 Approfondissement sur les sous-requêtes

Nous allons rebondir sur le chapitre précédent, et en particulier sur la dernière requête étudiée, pour approfondir la notion de sous-requête.

Revoyons notre requête au ralenti :

```
SELECT deptno, deptname,  
       (SELECT count(*) FROM employees WHERE deptno = workdept) as comptage  
FROM departements  
ORDER BY deptno
```

Je rappelle que le résultat obtenu est le suivant :

deptno	deptname	comptage
1	PLANNING	1
2	MANUFACTURING SYSTEMS	5
3	SOFTWARE SUPPORT	1
4	ADMINISTRATION SYSTEMS	0
5	OPERATIONS	3

Supposons que nous sommes contraints de conserver cette requête en l'état (on est à la bourre, trop de boulot, on n'a pas le temps de la réécrire), mais que nous souhaitons faire disparaître les départements n'ayant pas d'employés. La tentation serait grande d'écrire ceci :

```
SELECT deptno, deptname,  
       (SELECT count(*) FROM employees WHERE deptno = workdept) as comptage  
FROM departements  
WHERE comptage <> 0  
ORDER BY deptno
```

Mais cela ne fonctionne pas, SQL nous renvoie l'erreur suivante :

#1054 - Champ 'comptage' inconnu dans where clause

Il a raison le bougre, la colonne « comptage » n'existe pas au stade où nous essayons de l'utiliser. C'est une information qui sera renvoyée en sortie de la requête en cours, elle est produite par la requête en cours, mais elle n'est pas accessible dans le périmètre (en anglais le « scope ») de la requête en cours. Pour pouvoir exploiter cette colonne « comptage », il faudrait pouvoir nous placer à un niveau supérieur.

Nous pouvons nous placer à un niveau supérieur en encapsulant notre requête de manière à la transformer en sous-requête. Attention, on parle ici de sous-requête, mais pas de sous-requête

scalaire. Contrairement à une sous-requête scalaire, une sous-requête normale peut renvoyer plusieurs lignes et plusieurs colonnes.

Voici notre requête précédente, transformée en sous-requête d'une requête de niveau supérieure, que nous appellerons par commodité la requête parente. J'ai mis la requête parente en gras pour faciliter la lecture :

```
SELECT * FROM (  
    SELECT deptno, deptname,  
           (SELECT count(*) FROM employees WHERE deptno = workdept)  
           as comptage  
    FROM departements  
) as X  
WHERE X.comptage <> 0
```

Vous voyez donc que notre requête de tout à l'heure est devenue une sous-requête, encadrée par des parenthèses. Elle est exploitée par une clause FROM. Notre sous-requête se comporte donc comme une sorte de table temporaire, qui renvoie son jeu de données à la requête. Cette requête parente est en mesure d'exploiter les valeurs renvoyées par la colonne « comptage », nous avons donc la possibilité d'appliquer un filtre sur cette colonne avec la clause WHERE, comme dans l'exemple ci-dessus. Pour une meilleure lisibilité du code, j'ai mis un alias sur la requête « fille », alors que j'ai appelé « X ». Si vous êtes par la suite confronté à des requêtes embarquant plusieurs sous-requêtes, le fait d'attribuer un alias à chaque sous-requête vous permettra d'identifier instantanément l'origine de chaque donnée, vous gagnerez donc en confort pour la relecture et la maintenance du code.

Nous venons donc de voir un moyen simple et efficace d'éliminer les départements n'ayant pas d'employés affectés. Ce n'est sans doute pas aussi optimal que certaines techniques vues dans le chapitre précédent, mais cela fonctionne, et dans de nombreux cas c'est un très bon compromis. Sur certaines requêtes complexes, on n'a quelquefois pas d'autre choix que d'utiliser cette technique.

Nous sommes en décembre 2017, au moment où je rédige cette première version du support, et malheureusement la version de MariaDB embarquée dans XAMPP a un métré de retard par rapport à la dernière version de MariaDB disponible. XAMPP embarque une version 10.1.x, or MariaDB est maintenant disponible en version 10.2.x et propose une nouveauté extrêmement intéressante, la CTE (Common Table Expression).

La CTE existe depuis quelques temps sur d'autres SGBD comme PostgreSQL et DB2, et on l'attendait avec impatience sur MariaDB. Vous ne pourrez pas la tester avec votre version de MariaDB si vous êtes en version 10.1.x, mais voici concrètement comment la requête précédente pourrait s'écrire avec une CTE :

```
WITH tmp AS (  
    SELECT deptno, deptname,  
    (SELECT count(*) FROM employes WHERE deptno = workdept) as comptage  
    FROM departements  
)  
SELECT * FROM tmp  
WHERE tmp.comptage <> 0
```

Explication : la clause WITH permet d'indiquer à l'interpréteur SQL que nous allons utiliser une ou plusieurs CTE au sein de notre requête. Ici nous utiliserons une seule CTE que nous appelons « tmp ». Cette CTE est une sorte de table temporaire, préparée à l'avance, dont le contenu sera exploité par les CTE suivantes, ou la requête finale. Dans notre cas, nous n'avons pas d'autre CTE, c'est donc la requête finale qui exploite directement le jeu de données produit par la CTE « tmp ».

Concrètement, la requête ci-dessus produit strictement le même résultat que la requête précédente. Mais le code est structuré différemment, et dans le cas de requête complexes nécessitant l'utilisation de plusieurs sous-requêtes, la CTE est un formidable outil.

Pour information, si vous voulez combiner plusieurs CTE, il suffit de les séparer par des virgules, de façon suivante :

```
WITH  
cte1 as (SELECT * FROM xxx),  
cte2 as (SELECT x, y, z FROM cte1),  
cte3 as (SELECT count(*) FROM cte2)  
SELECT * FROM cte3
```

Dans l'exemple bidon ci-dessus, nous avons 3 CTE enchaînées (séparées par des virgules). Certaines de ces CTE exploitent des données produites par les CTE précédentes. Vous noterez que la requête finale n'est pas précédée d'une virgule, c'est le moyen qu'utilise SQL pour savoir qu'il s'agit de la requête finale qui produira le jeu de données définitif.

J'ai publié dans le numéro 211 de GNU Linux Magazine (de janvier 2018) un dossier présentant un certain nombre de nouveautés de MariaDB en version 10.2, dont les CTE et les fonctions de fenêtrage (Window Functions). Vous trouverez un lien vers ce magazine dans le chapitre « Bibliographie ».

3.4 Petite introduction aux jointures

Dans le chapitre précédent, nous avons étudié quelques techniques permettant d'effectuer un comptage des employés par département, et de récupérer dans le même temps le libellé des départements.

Il existe une autre technique qui consiste à effectuer une jointure entre les tables « employes » et « departements ». Il y a au moins 2 manières de réaliser cela, la méthode « historique » (désuète) et la méthode utilisant une jointure explicite (recommandée). Nous allons étudier ces 2 techniques.

Voici la méthode « historique », que je vous montre une fois, pour votre culture générale, mais que je ne montrerai plus par la suite, car elle est un peu désuète (même si elle fonctionne très bien):

```
SELECT workdept, deptname, count(*) as comptage
FROM employes, departements
WHERE deptno = workdept
GROUP BY workdept
```

workdept	deptname	comptage
1	PLANNING	1
2	MANUFACTURING SYSTEMS	5
3	SOFTWARE SUPPORT	1
5	OPERATIONS	3

On voit que les 2 tables sont définies toutes deux dans la clause FROM, séparées par une virgule, et que la condition de jointure entre ces 2 tables est définie dans la clause WHERE.

On peut dire que c'est relativement lisible, et compréhensible, sur une requête simple comme celle-ci, mais je vous laisse imaginer le bazar, quand vous devez lier plus de 2 tables par jointure, et que les critères se multiplient au sein de la clause WHERE. En effet, comment s'y retrouver entre les critères liés aux conditions de jointure et les critères de filtrage autres... le code peut très vite devenir confus avec cette technique de jointure.

Dans les toutes premières versions de la norme SQL, c'était la seule manière de faire (avec des variantes selon la manière dont la technique était implémentées dans les SGBD).

Voyons maintenant la méthode qui est recommandée aujourd'hui. Vous allez constater qu'elle est plus claire et plus évolutive, elle utilise une jointure explicite :

```
SELECT workdept, deptname, count(*) as comptage
FROM employes
INNER JOIN departements
  ON deptno = workdept
GROUP BY workdept
```

workdept	deptname	comptage
	1 PLANNING	1
	2 MANUFACTURING SYSTEMS	5
	3 SOFTWARE SUPPORT	1
	5 OPERATIONS	3

Le résultat est strictement le même, mais la jointure est décrite explicitement, avec la clause JOIN précédée de la clause INNER, et vous noterez que la clause WHERE précédente est devenue une clause « ON ». La condition de jointure reste cependant la même, avec une égalité sur les colonnes « deptno » et « workdept ».

La clause INNER indique que nous souhaitons sélectionner uniquement les lignes qui sont en stricte correspondance entre les 2 tables « employes » et « departements ».

Si nous avions souhaité obtenir la liste de tous les employés, avec à la fois ceux qui sont liés à un département, et ceux qui ne sont rattachés à aucun département, nous aurions pu utiliser la clause « LEFT OUTER JOIN », comme ceci :

```
SELECT workdept, deptname, count(*) as comptage
FROM employes
LEFT OUTER JOIN departements
  ON deptno = workdept
GROUP BY workdept
```

Dans le cas présent, le résultat obtenu est le même, car nous n'avons pas dans notre table « employes », d'employés rattachés à des départements inexistants. Mais c'est un cas qui pourrait se présenter dans la vraie vie, sur une base de données un peu vieillotte et pas très bien gérée (ce qui est le cas d'un grand nombre de bases de données en entreprise).

Testons ce cas de figure, en introduisant un employé fictif rattaché à un département qui n'existe pas :

```
INSERT INTO employes (EMPNO, LASTNAME, WORKDEPT)
VALUES (20, 'ORPHELIN', 10)
```

Et relançons la requête avec le LEFT OUTER JOIN :

workdept	deptname	comptage
1	PLANNING	1
2	MANUFACTURING SYSTEMS	5
3	SOFTWARE SUPPORT	1
5	OPERATIONS	3
10	<null>	1

On voit maintenant apparaître un département n° 10, avec un intitulé contenant la valeur « null », et un

Attention, ce genre de requête peut très vite devenir un enfer à maintenir, car quand on a beaucoup de tables en jointure, on se perd très vite dans la liste des colonnes, et on a du mal à savoir de quelles tables proviennent chacune des colonnes. Pour éviter cela, je vous recommande d'attribuer un alias à chaque table, comme dans l'exemple ci-dessous (qui est équivalent à celui de la page précédente) :

```
SELECT a.workdept, b.deptname, count(*) as comptage
FROM employes a
INNER JOIN departements b
  ON b.deptno = a.workdept
GROUP BY a.workdept
```

Le résultat obtenu est strictement le même mais la table « employes » s'appelle maintenant la table « a », la table « departements » s'appelle la table « b », et chaque colonne est préfixée avec le nom de sa table d'origine. Ainsi on sait précisément d'où vient chaque donnée, cela rend toute modification plus facile.

Une autre bonne pratique consiste à utiliser comme alias les préfixes des noms des tables (ou un code mnémotechnique se référant au nom des tables d'origine). Le code ci-dessous est équivalent au code précédent, mais l'alias « a » est devenu « emp » et l'alias « b » est devenu « dep ».

```
SELECT emp.workdept, dep.deptname, count(*) as comptage
FROM employes emp
INNER JOIN departements dep
  ON dep.deptno = emp.workdept
GROUP BY emp.workdept
```

Si l'on est amené à complexifier la requête ci-dessus, en ajoutant d'autres jointures pour répondre aux besoins des utilisateurs, et si l'on prend soin d'attribuer un alias à chaque table, alors la lecture et la maintenance de la requête seront plus faciles.

Nous avons vu beaucoup de choses dans ce chapitre autour des requêtes de type SELECT, avec en prime une petite initiation aux sous-requêtes scalaires et aux jointures.

Petite synthèse sur les principaux types de jointure :

- Le **INNER JOIN** permet de faire une jointure entre deux tables et de ramener les enregistrements de la première table qui ont une correspondance dans la seconde table.
- Le **LEFT OUTER JOIN** entre deux tables retourne tous les enregistrements ramenés par un **INNER JOIN** plus chaque enregistrement de la table 1 qui n'a pas de correspondance dans la table 2.
- Le **RIGHT OUTER JOIN** entre deux tables retourne tous les enregistrements ramenés par un **INNER JOIN** plus chaque enregistrement de la table 2 qui n'a pas de correspondance dans la table 1.

INNER JOIN et LEFT OUTER JOIN sont de très loin les techniques de jointure les plus utilisées. Mais MariaDB propose quelques variantes que vous pourrez découvrir par la suite. Pour une présentation plus complète du sujet, en rapport avec les possibilités de MariaDB :

<https://mariadb.com/kb/en/library/join-syntax/>

3.5 Petite introduction aux unions

On se sert quelquefois de la clause UNION pour lier ensemble des jeux de données provenant de plusieurs requêtes SQL.

D'un point de vue technique, cela donne ceci :

```
SELECT col1a, col1b, col1c from table1
UNION
SELECT col2a, col2b, col2c from table2
```

Les 2 requêtes SQL liées par UNION fonctionnent strictement sur le même principe que les requêtes que vous avez étudiées précédemment. Elles peuvent bien évidemment être beaucoup plus complexes que cela, avec des jointures, des clauses WHERE, GROUP BY, etc.

La principale contrainte à respecter, c'est que les jeux de données renvoyés par les 2 requêtes doivent être compatibles. Supposons en effet que la colonne « col1a » renvoyé par la première requête soit de type numérique, et que la colonne « col2a » renvoyé par la seconde requête soit de type alphanumérique, eh bien SQL renverrait un message d'erreur nous indiquant que les 2 requêtes liées par UNION ne sont pas compatibles.

Au niveau du nom des colonnes renvoyées par les 2 requêtes, c'est la « première qui a parlé qui a raison », donc le jeu de données renvoyé par la requête ci-dessus contiendra 3 colonnes nommées ainsi : col1a, col1b, col1c

Il faut souligner que, si les 2 requêtes renvoient des données identiques, la clause UNION éliminera les doublons. Si on avait souhaité voir apparaître ces doublons dans le jeu de données produit, on ajouterait la clause « ALL » derrière la clause UNION, comme ceci :

```
SELECT col1a, col1b, col1c from table1
UNION ALL
SELECT col2a, col2b, col2c from table2
```

Je vous proposerai un exemple d'utilisation de cette technique dans un prochain chapitre (le 3.6.3).

Quelques liens pour approfondir le sujet, si besoin :

<http://sql.sh/cours/union>

https://www.w3schools.com/sql/sql_union.asp

3.6 Liste des fonctions disponibles

Je vous propose de faire une pause, et d'en profiter pour faire le « tour du propriétaire ». Car MariaDB met à notre disposition un grand nombre de fonctions très pratiques.

3.6.1 Les fonctions associées aux chaînes de caractère

Nous allons tester à la volée un grand nombre de fonctions, au moyen de requêtes de ce type :

```
SELECT ascii('a');
```

Cela peut surprendre, surtout si vous connaissez déjà d'autres bases de données, car MariaDB (et bien sûr MySQL) n'impose pas de clause FROM dans l'écriture des requêtes. Sur la plupart des autres SGBD, nous aurions dû au moins utiliser une table pivot, comme ceci :

```
SELECT ascii('a') FROM pivot;
```

Voici un tableau (non exhaustif) présentant un certain nombre de fonctions de manipulation de chaînes :

Exemple	Résultat	Commentaire
SELECT ascii('a')	97	renvoie le code ASCII d'une lettre
SELECT bin(5)	101	renvoie la valeur binaire d'une chaîne
SELECT length('Hello')	5	renvoie la longueur d'une chaîne
SELECT char_length('Hello')	5	idem
SELECT bit_length('Hello')	40	renvoie la longueur d'une chaîne en fonction de l'encodage (ici UTF-8)
SELECT CONCAT('titi', 'gros minet')	titigros minet	concaténation sans séparateur
SELECT CONCAT_WS('-', 'titi', 'gros minet', 'zorro')	titi-gros minet-zorro	concaténation avec ajout de séparateur
SELECT FIELD('Carlson', 'Niki', 'xx', 'Carlson')	3	indique dans quelle colonne se trouve la valeur recherchée
SELECT FIND_IN_SET('Doe', 'John,Doe,Stas ie,Smith') as POSITION;	2	renvoie la position du mot recherché dans la chaîne spécifiée
select Format(1234.5555, 2)	1,234.56	arrondi de la valeur transmise selon nombre de décimales indiqué
SELECT INSERT('original', 2, 2, 'NEW')	oNEWginal	insère un mot à la position spécifiée
SELECT LOCATE('N', 'DINING')	3	renvoie la première position trouvée pour un caractère donné
SELECT LCASE('WORK'), UCASE('JCB')	work JOB	casse en minuscule ou majuscule
SELECT LOWER('WORK'), UPPER('JCB')	work JOB	idem
select substring('abcdef', 2, 3)	bcd	
SELECT STRCMP('testing', 'testing')	0	compare la longueur des 2 chaînes (1, 0, -1)
SELECT REVERSE('abc')	cba	
SELECT RIGHT('Inventory', 4)	tory	
SELECT RTRIM('PEAR ')	PEAR	
SELECT LTRIM(' PEAR')	PEAR	
SELECT TRIM(' PEAR ')	PEAR	
SELECT SOUNDEX('apple')	A140	
SELECT space(30)		

3.6.2 Les fonctions associées aux dates

Voici un tableau non exhaustif regroupant un certain nombre de fonctions de manipulation de date :

Syntaxe	Commentaire
ADDDATE(), DATE_ADD(), DATE_SUB(), SUBDATE()	ajout ou suppression de date
CURDATE(), CURRENT_DATE(), CURRENT_DATE	renvoie la date système
CURRENT_TIME(), CURRENT_TIME, CURTIME()	renvoie l'heure système
DAYOFMONTH(curdate())	renvoie le numéro de jour dans le mois
DAYOFWEEK(curdate())	renvoie le numéro de jour dans la semaine avec dimanche = jour 1
DAYOFYEAR(curdate())	renvoie le numéro de jour dans l'année
EXTRACT(YEAR_MONTH from curdate())	renvoie valeur numérique contenant année et mois
HOUR('10:05:03')	Renvoie « 10 »
DATE_FORMAT	(* voir ci-dessous)
STR_TO_DATE('15,12,2017','%d,%m,%Y')	Renvoie une date au format 2017-12-15
WEEK(curdate())	renvoie le numéro de semaine
WEEKDAY(curdate())	renvoie le numéro de jour dans la semaine avec lundi = jour 1

Avertissement : Dans les exemples du tableau, j'utilise souvent le mot clé « current_date » qui permet de récupérer la date système en SQL.

La fonction DATE_FORMAT est très pratique pour formater une date comme bon nous semble.

Exemples :

```
SELECT DATE_FORMAT(current_date, '%W, %M %D, %Y')
```

Résultat obtenu : Friday, December 8th, 2017

```
DATE_FORMAT('2017-12-08 10:23:00', '%Y-%M-%d %Hh%imin%ssec')
```

Résultat obtenu : 2017-December-08 10h23min00sec

Pour une présentation exhaustive des paramètres de la fonction DATE_FORMAT :

https://mariadb.com/kb/en/library/date_format/

On notera que la fonction DATE_FORMAT de MariaDB ressemble beaucoup à la fonction strftime() du langage PHP :

<http://php.net/manual/en/function.strftime.php>

3.6.3 Les fonctions d'agrégation

MariaDB fournit un certain nombre de fonctions d'agrégation.

Fonction	Désignation
AVG	Returns the average value
BIT_AND	Bitwise AND
BIT_OR	Bitwise OR
BIT_XOR	Bitwise XOR
COUNT	Returns count of non-null values.
COUNT DISTINCT	Returns count of number of different non-NULL values.
GROUP_CONCAT	Returns string with concatenated values from a group.
MAX	Returns the maximum value
MIN	Returns the minimum value
STD	Population standard deviation
STDDEV	Population standard deviation
STDDEV_POP	Returns the population standard deviation
STDDEV_SAMP	Standard deviation
SUM	Sum total
VARIANCE	Population standard variance
VAR_POP	Population standard variance
VAR_SAMP	Returns the sample variance

Dans le tableau ci-dessus, j'ai indiqué en gras celles des fonctions qui étaient le plus fréquemment utilisées.

Voici quelques exemples d'utilisation :

- obtenir le salaire minimum, maximum et moyen pour l'ensemble des salariés

```
SELECT
    min(montant) as sal_min,
    max(montant) as sal_max,
    avg(montant) as sal_avg
FROM salaires
```

- compter le nombre total de salariés :

```
SELECT count(*) as nb_total
FROM employes
```

- compter le nombre total de salariés par département (déjà évoqué précédemment):

```
SELECT workdept, count(*) as tot_emp
FROM employes
GROUP BY workdept
```

- obtenir la somme des salaires par département :

```
SELECT employes.WORKDEPT, sum(salaires.montant) as tot_sal
FROM employes
INNER JOIN salaires
  ON employes.EMPNO = salaires.EMPNO
GROUP BY employes.WORKDEPT
```

C'est pas mal, mais c'est mieux si nous ajoutons le libellé des départements, comme ceci :

```
SELECT emp.WORKDEPT as deptno,
       dep.deptname,
       sum(sal.montant) as tot_sal
FROM employes emp
INNER JOIN salaires sal
  ON emp.EMPNO = sal.EMPNO
INNER JOIN departements dep
  ON dep.deptno = emp.workdept
GROUP BY emp.WORKDEPT, dep.deptname
```

Jetons un coup d'oeil au résultat :

deptno	deptname	tot_sal
1	PLANNING	45000
2	MANUFACTURING SYSTEMS	236571
3	SOFTWARE SUPPORT	41000
5	OPERATIONS	128947

C'est cool. Mais vous vous souvenez peut être que nous avons parlé précédemment de la clause UNION. Je vous avais dit que je trouverais un exemple d'utilisation dans le présent chapitre. Voici l'exemple en question, qui consiste à réutiliser la requête que nous venons d'étudier, et à lui ajouter une requête totalisant les salaires tous départements confondus. Les 2 requêtes sont bien évidemment liées par la clause UNION. Voici la requête :

```
SELECT emp.WORKDEPT as deptno,
       dep.deptname,
       sum(sal.montant) as tot_sal
FROM employes emp
INNER JOIN salaires sal
  ON emp.EMPNO = sal.EMPNO
INNER JOIN departements dep
  ON dep.deptno = emp.workdept
GROUP BY emp.WORKDEPT, dep.deptname
UNION
SELECT NULL, 'TOTAL SALAIRES : ', SUM(montant) FROM salaires
```

Et voici bien évidemment le résultat :

deptno	deptname	tot_sal
1	PLANNING	45000
2	MANUFACTURING SYSTEMS	236571
3	SOFTWARE SUPPORT	41000
5	OPERATIONS	128947
	TOTAL SALAIRES :	451518

3.6.3 Les fonctions associées aux nombres

MariaDB fournit un certain nombre de fonctions numériques et trigonométriques.

Voici un tableau récapitulant les principales fonctions disponibles. J'ai mis en gras les fonctions qui sont d'un usage plus fréquent que les autres, notamment pour le développement d'applications de gestion.

Je vous encourage à tester ces fonctions pour vous familiariser avec leur utilisation.

Fonction	Désignation
ABS	Returns an absolute value
ACOS	Returns an arc cosine
ASIN	Returns the arc sine
ATAN	Returns the arc tangent
ATAN2	Returns the arc tangent of two variables
CEIL	Synonym for CEILING()
CEILING	Returns the smallest integer not less than X
CONV	Converts numbers between different number bases
COS	Returns the cosine
COT	Returns the cotangent
CRC32	Computes a cyclic redundancy check value
DEGREES	Converts from radians to degrees
EXP	e raised to the power of the argument
FLOOR	Largest integer value not greater than the argument
GREATEST	Returns the largest argument
LEAST	Returns the smallest argument
LN	Returns natural logarithm
LOG	Returns the natural logarithm
LOG10	Returns the base-10 logarithm
LOG2	Returns the base-2 logarithm
MOD	Modulo operation. Remainder of N divided by M
OCT	Returns octal value
PI	Returns the value of π (pi)
POW	Returns X raised to the power of Y
POWER	Synonym for POW()
RADIANS	Converts from degrees to radians
RAND	Random floating-point value.
ROUND	Rounds a number
SIGN	Returns 1, 0 or -1
SIN	Returns the sine
SQRT	Square root
TAN	Returns the tangent
TRUNCATE	Truncates X to D decimal places

Pour une présentation détaillée de ces différentes fonctions :

<https://mariadb.com/kb/en/library/numeric-functions/>

3.7 Les requêtes d'insertion, de mise à jour et de suppression

3.7.1 Les requêtes INSERT

Nous avons étudié pas mal de requêtes de type SELECT, nous avons aussi vu quelques exemples de requêtes INSERT. Nous allons revenir rapidement sur les requêtes INSERT, et découvrir également les requêtes de mise à jour (UPDATE) et de suppression (DELETE).

Vous vous souvenez sans doute que notre table des employés contient les 3 colonnes « empno », « lastname » et « workdept ».

Voici un exemple d'insertion de requête INSERT :

```
INSERT INTO employees (empno, lastname, workdept) VALUES (10, 'TEST', 20);
```

Nous pouvons écrire aussi ceci, qui est strictement équivalent :

```
INSERT INTO employees VALUES (10, 'TEST', 20);
```

Cette seconde requête fonctionne car nous avons respecté l'ordre des colonnes de la table et aussi le nombre de ces colonnes. Pour l'instant la table ne contient que 3 colonnes, mais supposons qu'un jour un développeur modifie la structure de la table et y ajoute une 4ème colonne. Dans ce cas de figure, la première requête continuerait à fonctionner, car les colonnes destinataires de l'INSERT sont clairement indiquées (avant le mot clé VALUES). Dans le cas de la seconde requête en revanche, nous aurions un plantage car SQL ne saurait pas faire le lien entre les 3 valeurs insérées (10, 'TEST' et 20) et les 4 colonnes de la table modifiée. C'est la raison pour laquelle je recommande de toujours utiliser la première requête plutôt que la seconde, cela permet de garantir un code plus robuste, moins dépendant des modifications de la base de données.

Nous avons vu dans un précédent chapitre qu'il était possible d'alimenter une table à partir d'un SELECT. En supposant que notre table des employés est alimentée par un jeu de données récupéré d'une ancienne table appelée « old_empl », cela pourrait donner ceci :

```
INSERT INTO employees (empno, lastname, workdept)
```

```
SELECT col1, col2, col3 FROM old_empl WHERE col1 > 0 and col3 > 0
```

Bien évidemment la requête SELECT utilisée pour l'insertion peut être beaucoup plus complexe que celle ci-dessus, et utiliser l'ensemble des techniques de jointure que nous avons étudiées précédemment.

3.7.2 Les requêtes UPDATE

Passons maintenant aux requêtes de mise à jour. Voici un exemple tout simple, dans lequel nous mettons à jour le nom et le département du salarié qui porte le numéro 10 :

```
UPDATE employes SET lastname='TEST', workdept=5 WHERE empno=10
```

Il s'agit ici d'une mise à jour unitaire, mais la grande force de SQL, c'est de permettre la modification d'un grand nombre de ligne – voire d'un très grand nombre de lignes – très rapidement. Notre table des salaires est toute petite, mais voici un exemple dans lequel nous décidons d'augmenter de 10 % les salaires de l'ensemble des salariés :

```
UPDATE salaires SET montant = montant * 1.1
```

Nous pouvons aussi souhaiter augmenter de 5 % les salaires travaillant dans le département n° 2. Aïe, ça devient plus compliqué car la notion de département ne figure pas dans la table des salaires. Avant de nous pencher sur la requête de mise à jour finale, regardons de quelle manière nous pouvons déterminer la liste des salaires qui seront impactés par notre requête de mise à jour. Pour cela, commençons par écrire la requête SELECT correspondant à notre recherche :

```
SELECT * FROM salaires WHERE empno IN  
  (SELECT empno FROM employes WHERE workdept = 2)
```

Nous avons déjà vu la clause IN et nous avons vu qu'elle pouvait recevoir une série de valeurs codées en dur, comme par exemple IN(2, 3, 5). Mais nous pouvons utiliser aussi la clause IN pour tester les valeurs renvoyées par une sous-requête de type SELECT, et c'est ce que nous venons d'écrire dans le code ci-dessus. Pratique non ?

La requête actuelle, compte tenu de notre jeu de données, renvoie 5 lignes correspondant aux salaires de 5 salariés différents. Si nous modifions la requête en inversant la clause WHERE de la sous-requête, comme ceci :

```
SELECT * FROM salaires WHERE empno IN  
  (SELECT empno FROM employes WHERE workdept <> 2)
```

... nous obtenons une liste de salariés qui n'a rien à voir avec la liste précédente.

Donc notre requête de mise à jour finale ne devrait impacter que 5 lignes de la table des salaires. Voici cette requête :

```
UPDATE salaires SET montant = montant * 1.05  
WHERE empno IN (SELECT empno FROM employes WHERE workdept = 2)
```

Je vous invite à tester ce cas de figure, et à y passer du temps pour bien comprendre le principe.

Vous avez vu que j'ai commencé par réaliser une requête de type SELECT me permettant de déterminer la meilleure manière d'accéder aux lignes à mettre à jour. Et c'est seulement ensuite que je me suis attelé à l'écriture de la requête de mise à jour, en réutilisant l'essentiel de la requête précédente (et en particulier la sous-requête). C'est une bonne méthode de travail, que je vous recommande d'adopter. Cela vous permettra de travailler avec le maximum de sécurité, particulièrement si vous devez intervenir sur une base de données pour y corriger des anomalies (ce qui arrive plus souvent que vous ne pouvez l'imaginer).

3.7.3 Les requêtes DELETE

Si on a besoin de vider complètement une table de ses données, certains SGBD fournissent l'instruction TRUNCATE, c'est le cas de MariaDB. Si par exemple vous voulez vider complètement la table des employés, vous pourrez écrire ceci :

```
TRUNCATE TABLE employes;
```

Vous auriez obtenu presque le même résultat en écrivant ceci :

```
DELETE FROM employes;
```

Je dis bien « presque » car en règle générale l'ordre TRUNCATE est plus rapide que le DELETE. Mais quand on souhaite ne supprimer que quelques lignes, on doit passer par DELETE, et utiliser un filtre avec la clause WHERE, comme ceci :

```
DELETE FROM employes WHERE empno IN (2, 10);
```

Vous l'aurez compris, on supprime ici les lignes pour lesquelles la colonne « empno » contient les valeurs 2 ou 10. Si aucune ligne n'est concernée par cette sélection, le moteur SQL se contentera d'indiquer qu'il a modifié 0 lignes.

Supposons maintenant que l'on souhaite supprimer les salaires des salariés des salariés travaillant dans le département 3 (drôle d'idée, mais c'est pour l'exemple), nous pouvons nous baser sur l'exemple de requête du chapitre précédent :

```
SELECT * FROM salaires WHERE empno IN  
  (SELECT empno FROM employes WHERE workdept = 3)
```

... et nous pouvons réutiliser ce matériel pour créer notre requête de suppression, comme ceci :

```
DELETE FROM salaires WHERE empno IN  
  (SELECT empno FROM employes WHERE workdept = 3)
```

Si votre jeu de données est identique au mien, alors vous venez de supprimer une ligne de la table des salaires.

Nous verrons dans un des chapitre qui suivent comment corriger certains anomalies que l'on peut rencontrer dans une base de données mal gérée, et nous verrons notamment comment supprimer des doublons.

3.8 Les astuces des pros

3.8.1 Identifier les doublons et les éliminer

Vous serez peut être confronté un jour au problème suivant : on vous signale un beau matin qu'il y a des doublons dans une table SQL, et il faut que vous trouviez une solution pour les éliminer, et de préférence... rapidement !

C'est en effet une situation qui peut se produire, souvent provoqué par un traitement mal écrit, ou quelquefois provoqué par une mauvaise manipulation. Quoi qu'il en soit, nous devons trouver une solution pour identifier les doublons.

Supposons que le problème la table des salaires, et supposons que toutes les lignes de la table soient « doublonnées », comme on dit vulgairement. On peut provoquer volontairement cette situation en utilisant la requête suivante :

```
INSERT INTO salaires (empno, montant)
SELECT empno, montant FROM salaires
```

Vous pouvez vérifier après ça – via un simple SELECT - que vous avez bien l'ensemble des lignes doublées.

Sur certaines bases de données, comme par exemple DB2, pour éliminer les doublons, nous aurions pu écrire ceci :

```
DELETE FROM salaires A
WHERE RRN(A) NOT IN (
    SELECT MAX( RRN(B) )
    FROM salaires B
    WHERE A.empno = B.empno AND A.montant = B.montant
)
```

Cela aurait fonctionné avec DB2 car cette base de données fournit une fonction RRN() qui a le grand avantage de nous fournir le « relative record number », soit le numéro relatif à chaque ligne. Ce numéro est un identifiant unique fourni pour chaque ligne par DB2. Cet identifiant unique ne change pas si on relance la requête plusieurs fois. Malheureusement cette fonction RRN() n'existe pas sur MariaDB.

Si on était sur PostgreSQL (en version 9) ou sur MariaDB en version 10.2, on pourrait exploiter des fonctions « Window » telles que ROW_NUMBER() et OVER() qui permettraient de récupérer un numéro unique pour chaque ligne (cf. lien ci-dessous pour de plus amples précisions) :

https://mariadb.com/kb/en/library/row_number/

Malheureusement, nous sommes en décembre 2017, et à l'heure où j'écris ces lignes, MariaDB est fourni sur de nombreux stacks PHP en version 10.1.x, version qui ne dispose pas de ces nouvelles fonctions « Window ».

Mais tout n'est pas perdu, car on peut recourir à la technique suivante, que j'ai trouvée dans certains forums et que j'ai testée pour vous (je confirme que ça fonctionne) :

```
ALTER IGNORE TABLE salaires ADD UNIQUE INDEX tmpuniq (empno, montant);
```

on recommandera de supprimer l'index « tmpuniq » par la suite (via un « DROP INDEX »), sauf si l'on souhaite empêcher toute nouvelle injection de doublon (auquel cas on aura intérêt à conserver cet index).

Bon, une autre technique un peu « bourin » mais néanmoins efficace aurait consisté à créer à une table temporaire contenant un jeu de données dédoublonné (grâce à la clause DISTINCT qui a pour effet d'éliminer les doublons), puis de vider la table des salaires et de la réalimenter la table des salaires à partir de la table temporaire. Concrètement, cela aurait donné ceci:

```
-- création d'une enveloppe vide
CREATE TABLE tmpsalaires
SELECT * FROM salaires limit 0;

-- insertion des données dédoublonnées grâce à la clause DISTINCT
INSERT INTO tmpsalaires (empno, montant)
SELECT DISTINCT empno, montant FROM salaires;

-- vidage de la table salaires
DELETE FROM salaires ;

-- réinjection des données
INSERT INTO salaires SELECT * FROM tmpsalaires ;
```

On voit qu'en SQL il y souvent plusieurs manières de faire les choses, et surtout que l'on est parfois contraint par les limites techniques de certains SGBD. Le fait que MySQL (et MariaDB avant la version 10.2) ne fournisse pas de fonction pratique renvoyant le numéro relatif de chaque ligne est une sacrée contrainte, qui nous oblige à quelques contorsions plus ou moins élégantes, pour nous en sortir.

3.8.2 Identifier les orphelins et les éliminer

Suite à une anomalie dans un traitement (peut être provoqué par une application PHP mal fichue), vous vous apercevez la table des employés contient des employés rattachés à des départements qui n'existent pas, ou plus.

On parle souvent d'enregistrements orphelins, ou de ligne orphelines, pour désigner ce genre de situation, qui arrive souvent sur des bases de données dans lesquelles des contraintes d'intégrité référentielle n'ont pas été mises en place. Cela peut s'expliquer par exemple par le fait que le moteur MyISAM de MySQL ne sait pas gérer ce type de contrainte (contrairement au moteur InnoDB), ou par le fait la gestion de contraintes d'intégrité référentielle est un travail « pointu » qui implique un véritable travail d'administration de la base de données (or beaucoup de sociétés rechignent à embaucher un DBA à plein temps, et c'est d'ailleurs une compétence qui n'est pas si évidente à trouver).

Quoi qu'il en soit, vous avez des lignes orphelines dans la table des employés. Pour les identifier, vous pouvez utiliser la requête suivante :

```
SELECT * from employes WHERE workdept NOT IN (
    SELECT deptno FROM departements WHERE deptno = workdept
)
```

Si la requête ne vous renvoie rien, c'est que votre table est « clean ». Nous allons donc tricher un peu en introduisant 2 lignes orphelines, une avec un employé pointant sur un département inexistant, et l'autre avec un employé dont le code département est à nul :

```
INSERT INTO employes (empno, lastname, workdept)
VALUES
(40, 'orphelin n° 1', 999),
(41, 'orphelin n° 2', null)
```

Si vous relancez la requête SELECT, les 2 lignes orphelines doivent maintenant apparaître. Renseignement pris auprès du services des ressources humaines (RH), les 2 orphelins correspondent à des employés qui ne sont plus dans la société depuis longtemps, et que l'on peut retirer de la base.

Voici une requête relativement simple, dérivée de la requête SELECT précédente, qui va nous permettre de nettoyer la table des employés :

```
DELETE FROM employes WHERE workdept NOT IN (
    SELECT deptno FROM departements WHERE deptno = workdept
)
```

Et voilà, 2 lignes supprimées, tout va bien. C'est vrai que l'on était sur un exemple scolaire, mais dans la vraie vie, il se pourrait que les 2 employés supprimés soient référencés dans d'autres tables, comme par exemple la table des salaires, et quelques autres... La suppression de lignes orphelines doit donc s'accompagner de précautions, et il est généralement préférable de ne pas traiter le cas dans l'urgence. Il vaut mieux en effet effectuer un audit de la base, pour mesurer les impacts d'une suppression de ligne. Si

on oublie des dépendances, on risque en supprimant des lignes orphelines, d'en créer de nouvelles dans d'autres tables.

3.8.3 Faire pivoter des données

Nous allons étudier dans ce chapitre des techniques d'inversion de données, permettant de mettre des lignes en colonnes, ou des colonnes en lignes.

Pour l'écriture de ce chapitre, je me suis très largement inspiré de techniques trouvées dans le livre suivant :

"SQL Hacks", d'Andrew Cumming et Gordon Russell, éd. O'Reilly

On retrouve ce livre dans les références bibliographiques qui se trouvent à la fin de manuel.

Je trouve ces techniques très intéressantes car elles permettent de découvrir un aspect méconnu de SQL, et de mieux percevoir l'étendue de ses possibilités.

Comment mettre des lignes en colonnes

Pour commencer, créons une table des notes.

```
CREATE TABLE notes
(ETUDIANT CHAR (20 ),
 COURS     CHAR (20 ),
 NOTE      INTEGER );

INSERT INTO notes (etudiant, cours, note) VALUES
('Gao Gong', 'Java', 80),
('Gao Gong', 'BD', 77),
('Gao Gong', 'Algèbre', 50),
('Zang Yi', 'Java', 62),
('Zang Yi', 'BD', 95),
('Zang Yi', 'Algèbre', 63);

SELECT * FROM notes;
```

ETUDIANT	COURS	NOTE
Gao Gong	Java	80
Gao Gong	BD	77
Gao Gong	Algèbre	50
Zang Yi	Java	62
Zang Yi	BD	95
Zang Yi	Algèbre	63

Pour la suite de l'exercice, on a besoin d'une table contenant la liste des étudiants. Comme nous n'en avons pas, je vous propose d'en créer une artificiellement via la requête suivante :

```
CREATE TABLE etudiants AS SELECT DISTINCT etudiant FROM notes
```

```
SELECT * FROM etudiants ;
```

ETUDIANT

Gao Gong

Zang Yi

Pour mettre les lignes en colonnes, nous disposons de 2 solutions.

1^{ère} solution : utiliser 3 jointures externes entre la table « etudiants » et la table « notes » :

```
SELECT a.etudiant, jav.note as java, bas.note as bd, alg.note as algebre
FROM etudiants a
LEFT OUTER JOIN notes jav
ON (a.etudiant = jav.etudiant and jav.cours = 'Java')
LEFT OUTER JOIN notes bas
ON (a.etudiant = bas.etudiant and bas.cours = 'BD')
LEFT OUTER JOIN notes alg
ON (a.etudiant = alg.etudiant and alg.cours = 'Algèbre');
```

ETUDIANT	JAVA	BD	ALGEBRE
Gao Gong	80	77	50
Zang Yi	62	95	63

2^{ème} solution : utiliser une seule jointure avec l'instruction CASE, et un GROUP BY sur le nom de l'étudiant.

```
SELECT a.etudiant,
max(case when b.cours = 'Java' then b.note else 0 end) as java,
max(case when b.cours = 'BD' then b.note else 0 end) as BD,
max(case when b.cours = 'Algèbre' then b.note else 0 end) as Algebre
FROM etudiants a
LEFT OUTER JOIN notes b
ON (a.etudiant = b.etudiant)
GROUP BY a.etudiant ;
```

ETUDIANT	JAVA	BD	ALGEBRE
Gao Gong	80	77	50
Zang Yi	62	95	63

N.B. : sans la clause GROUP BY, on n'aurait pas pu utiliser la clause MAX sur les colonnes calculées « Java », « BD » et « Algebre ». On aurait donc eu une requête telle que celle-ci-dessous :

```
select a.etudiant,
```



```

case when b.cours = 'Java' then b.note else 0 end as Java,
case when b.cours = 'BD' then b.note else 0 end as BD,
case when b.cours = 'Algèbre' then b.note else 0 end as Algebre
from etudiants a
left outer join notes b on (a.etudiant = b.etudiant);

```

qui aurait abouti au résultat suivant :

ETUDIANT	JAVA	BD	ALGEBRE
Gao Gong	80	0	0
Gao Gong	0	77	0
Gao Gong	0	0	50
Zang Yi	62	0	0
Zang Yi	0	95	0
Zang Yi	0	0	63

Comment mettre des colonnes en lignes

Pour aller plus vite, nous allons repartir du chapitre précédent, en créant une table “etudnotes” qui recevra le résultat de la requête créée précédemment.

```

CREATE TABLE etudnotes
(ETUDIANT CHAR (20 ),
JAVA INTEGER,
BD INTEGER,
ALGEBRE INTEGER);

```

```

INSERT INTO etudnotes
SELECT a.etudiant,
max(case when b.cours = 'Java' then b.note else 0 end) as Java,
max(case when b.cours = 'BD' then b.note else 0 end) as BD,
max(case when b.cours = 'Algèbre' then b.note else 0 end) as Algebre
FROM etudiants a
LEFT OUTER JOIN notes b ON (a.etudiant = b.etudiant)
GROUP BY a.etudiant ;

```

Nous obtenons donc la table suivante avec une présentation en colonnes :

ETUDIANT	JAVA	BD	ALGEBRE
Gao Gong	80	77	50
Zang Yi	62	95	63

... à partir de laquelle nous pouvons afficher les colonnes en ligne au moyen de la requête suivante :

```

SELECT etudiant, 'Java' as matiere, Java as note FROM etudnotes
UNION
SELECT etudiant, 'BD' as matiere, BD FROM etudnotes
UNION
SELECT etudiant, 'Algèbre' as matiere, Algebre FROM etudnotes ;

```

ETUDIANT	MATIERE	NOTE
Gao Gong	Java	80

Zang Yi	Java	62
Gao Gong	BD	77
Zang Yi	BD	95
Gao Gong	Algèbre	50
Zang Yi	Algèbre	63

Pour obtenir un tri par étudiant et matière, on peut écrire ceci :

```
SELECT x.* FROM (  
  SELECT etudiant, 'Java' as matiere, Java as note FROM etudnotes  
  UNION  
  SELECT etudiant, 'BD' as matiere, BD FROM etudnotes  
  UNION  
  SELECT etudiant, 'Algèbre' as matiere, Algebre FROM etudnotes  
) x ORDER BY x.etudiant, x.matiere  
;
```

3.8.4 Sous-requêtes scalaires, données temporelles

NB : le présent chapitre est repris presque intégralement d'un de mes autres supports de cours (consacré lui à PHP et MySQL). J'ai souhaité reporter ce chapitre ici, pour éviter aux personnes souhaitant utiliser MySQL (ou MariaDB) avec un autre langage serveur (par exemple NodeJS), de devoir se plonger dans un support dédié à PHP.

Je vous propose une petite étude de cas, largement inspirée d'un projet réel sur lequel j'ai travaillé il y a quelques années. Le projet en question consistait à développer une gestion de catalogue produit, pour une société spécialisée dans la revente de matériel agricole. L'activité de la société en réalité importe peu, car les techniques que nous allons étudier sont valables pour tout type d'activité, mais cela explique pourquoi le jeu d'essai que nous allons utiliser parle de motoculteur, de tronçonneuse, etc... Je l'ai bien évidemment dépersonnalisé, et largement simplifié, en ne gardant que les tables et colonnes qui me semblaient utiles pour notre étude de cas.

Un point très important également : les techniques SQL que je vais présenter dans ce chapitre fonctionnent sur de nombreuses bases de données. Je les ai pour ma part utilisées avec succès sur les SGBD (Systèmes de Gestion de Base de Données) MySQL, PostgreSQL et DB2. Je ne doute pas que vous pourrez les utiliser sur d'autres SGBD sans trop de modifications.

Nous n'allons pas développer ici toute la gestion de catalogue produit, nous allons plutôt développer un module de consultation proche de celui présenté au chapitre précédent, mais avec des requêtes SQL plus complexes, car les données que nous souhaitons afficher sont réparties dans plusieurs tables.

Dans notre étude de cas, nous disposons de 4 tables :

prd_produit : la table des produits, comme son nom l'indique

prd_famille : la table des familles de produits

prd_activite : la table des activités

prd_prixvte : la table des prix de vente

Nous avons quelques règles fonctionnelles à énoncer :

chaque produit est lié à une famille et une seule, mais une famille peut être associée à 0 ou N produits

chaque famille est liée à une activité et une seule, mais une activité peut être associée à 0 ou N familles

chaque prix de vente est lié à un produit et un seul

un produit peut avoir zéro ou N prix de vente, mais il ne peut avoir qu'un seul prix de vente à une date donnée

Prenons pour exemple la table des activités. Voici sa structure :

```
1 CREATE TABLE prd_activite (  
2   id int(10) unsigned NOT NULL AUTO_INCREMENT,  
3   nom varchar(80) DEFAULT NULL,  
4   PRIMARY KEY (id)  
5 ) ENGINE=InnoDB DEFAULT CHARSET=utf8 ;|
```

... suivi de son jeu de données :

```
INSERT INTO prd_activite (id, nom) VALUES  
(1, 'Motoculture'),  
(2, 'Remorques'),  
(3, 'Quads'),  
(4, 'Cycles'),  
(5, 'Matériel agricole et espace vert'),  
(6, 'Matériel de travaux publics');
```

Il s'agit d'une table vraiment très simple, avec 2 colonnes qui contiennent respectivement :

un identifiant numérique en incrémentation automatique,
un nom de 80 caractères de long (maxi),

Je vous disais que les techniques SQL que nous allons utiliser étaient quasiment identiques sur plusieurs SGBD, et c'est vrai. Mais dans la norme SQL, il y a 2 grandes parties qui sont :

DDL (Data Definition Language) : cela correspond au code SQL de création et de gestion des tables et autres objets SQL (vues, procédures stockées, etc...)

DML (Data Manipulation Language) : cela désigne le code SQL d'interrogation et de manipulation du **contenu** des tables, soit les requêtes SELECT, UPDATE, INSERT et DELETE, et toutes les techniques de jointure qui vont avec.

Les plus grosses différences syntaxiques entre SGBD se situent surtout au niveau de DDL, mais on trouve aussi des différences côté DML, notamment sur les fonctions (de conversion de type, de manipulation de dates, etc...), et aussi sur certaines techniques d'interrogation avancées qui ne fonctionnent que sur certains SGBD. Je pense par exemples aux techniques de type (Common Table Expression) qui fonctionnent à peu près de la même manière sur PostgreSQL et DB2, et n'existent tout simplement pas sur MySQL.

Jetons maintenant un coup d'œil à la structure des tables familles, produits et prix de vente.

```
1 CREATE TABLE prd_famille (  
2   id int(10) unsigned NOT NULL AUTO_INCREMENT,  
3   nom varchar(80) DEFAULT NULL,  
4   prd_activite_id int(10) unsigned NOT NULL,  
5   PRIMARY KEY (id)  
6 ) ENGINE=InnoDB DEFAULT CHARSET=utf8 ;
```

```
1 CREATE TABLE prd_produit (  
2   id int(10) UNSIGNED NOT NULL AUTO_INCREMENT,  
3   codeproduit char(50) NOT NULL,  
4   prd_famille_id int(10) UNSIGNED NOT NULL,  
5   PRIMARY KEY (id)  
6 ) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

```
1 CREATE TABLE prd_prixstd (  
2   id int(10) UNSIGNED NOT NULL AUTO_INCREMENT,  
3   prd_produit_id int(10) UNSIGNED NOT NULL,  
4   date_deb date NOT NULL,  
5   prix decimal(11,5) NOT NULL,  
6   PRIMARY KEY (id)  
7 ) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

Comme les jeux de données sont un peu plus volumineux pour ces tables, je les ai regroupés dans un chapitre de l'annexe (cf. chapitre 5.8).

Vous avez remarqué que chaque table a une colonne « id ». Cette colonne « id » est un identifiant numérique en incrémentation automatique, qui sert de clé primaire (cf. mot clé « PRIMARY KEY »). Chaque ligne d'une table a donc un identifiant unique qui la définit. Dans certaines applications professionnelles, il peut arriver que la clé primaire d'une table soit composée de plusieurs colonnes, on parle alors de « clé primaire composite ». Ce ne sera pas le cas dans notre application, je n'ai pas souhaité la compliquer plus que nécessaire.

Nous avons précisé tout à l'heure qu'une famille était liée à une activité et une seule. Techniquement cela se traduit par la présence d'une colonne « prd_activite_id » à l'intérieur de la table « prd_famille ». On dit que « prd_activite_id » est une « clé étrangère ».

Pour obtenir la liste des familles avec leurs activités respectives, nous pouvons écrire une jointure entre les tables « famille » et « activité », avec le code SQL suivant :

```
SELECT a.id, a.nom, a.prd_activite_id, b.nom  
FROM prd_famille a  
INNER JOIN prd_activite b
```

```
ON a.pr_d_activite_id = b.id
ORDER BY b.id, a.id ;
```

Voici un petit échantillon de ce que produit la requête :

id	nom	prd_activite_id	nom
26	Tracteurs de pelouses	1	Motoculture
27	Tronçonneuses à chaîne	1	Motoculture
28	Tronçonneuses d'élagage	1	Motoculture
29	Tronçonneuses sur perche	1	Motoculture
56	Pulvérisateur	1	Motoculture
61	Tondeuses robot	1	Motoculture
30	Accessoires	2	Remorques
31	Agricoles	2	Remorques
34	Utilitaires	2	Remorques
36	Sports (homologués route)	3	Quads

La clause « INNER JOIN » permet de sélectionner dans la jointure uniquement les lignes qui existent dans les 2 tables liées par la jointure. Si certaines familles sont liées à des activités qui n'existent pas (pour cause de suppression accidentelle), alors elles n'apparaîtront pas avec ce type de requête.

Des familles qui « pointent » sur des activités qui n'existent pas, on appelle ça des orphelins (ou des « lignes orphelines »), en jargon de professionnel (je vous garantis que ce n'est pas une blague).

Vous pourriez être tenté de me répondre que les orphelins ne pourraient pas exister si des contraintes d'intégrité référentielle avaient été définies au niveau SQL, et je vous répondrai que vous avez raison. Mais les contraintes d'intégrité référentielles ne sont que très rarement mises en œuvre dans les projets en entreprise, pour diverses raisons sur lesquelles je ne m'étendrai pas ici. Donc, dans la « vraie vie », des bases de données avec des données orphelines, je pense que vous allez en rencontrer souvent.

Si vous pensez que vous risquez d'avoir des familles orphelines de leurs activités, et si vous souhaitez les voir apparaître quand même, vous pouvez modifier légèrement la requête SQL en remplaçant la clause « INNER JOIN » par la clause « LEFT JOIN », comme ceci :

```
SELECT a.id, a.nom, a.pr_d_activite_id, b.nom
FROM prd_famille a
LEFT JOIN prd_activite b
ON a.pr_d_activite_id = b.id
ORDER BY b.id, a.id ;
```

Il y a une autre manière de lier les 2 tables famille et activité, c'est en utilisant une sous-requête scalaire. Cette technique est moins connue que la précédente, et dans certains cas elle est bien pratique, alors je vous montre ce que cela donne :

```
SELECT a.id, a.nom, a.pr_d_activite_id ,  
(SELECT b.nom FROM pr_d_activite b  
WHERE a.pr_d_activite_id = b.id) as nom_activite  
FROM `pr_d_famille` a  
ORDER BY nom_activite, a.id
```

On voit dans notre exemple que la sous-requête est définie dans la clause SELECT de la requête principale. Je l'ai « stabilisée » pour vous aider à la repérer, elle est encapsulée dans un jeu de parenthèses, et elle alimente le contenu d'une colonne générée dynamiquement qui s'appelle « nom_activite ».

La sous-requête scalaire a pour particularité de renvoyer une seule ligne et une seule colonne. S'il y a un risque que la sous-requête scalaire renvoie plus d'une ligne, on peut l'obliger à n'en renvoyer qu'une seule en utilisant la clause « limit 1 » comme dans l'exemple suivant :

```
SELECT a.id, a.nom, a.pr_d_activite_id ,  
(SELECT b.nom FROM pr_d_activite b  
WHERE a.pr_d_activite_id = b.id LIMIT 1) as nom_activite  
FROM `pr_d_famille` a  
ORDER BY nom_activite, a.id
```

Bon, dans notre exemple, la clause « LIMIT 1 » est inutile (de par la nature de notre jeu d'essai), mais vous pourrez trouver d'autres cas où cette technique peut rendre service.

Nous avons utilisé la sous-requête scalaire à l'intérieur de la clause SELECT, mais on peut l'utiliser pour récupérer une information que l'on souhaite comparer avec une donnée de la requête principale, et dans ce cas on placera la sous-requête scalaire à l'intérieur de la clause WHERE. Nous verrons une mise en application dans un instant.

Intéressons-nous à la table des produits, et voyons comment nous pouvons lier ensemble différentes tables. Si je souhaite obtenir la liste des produits, triés par famille et code produit, avec en affichage les libellés activité et famille, suivi du code produit, je peux écrire ceci :

```

SELECT a.nom as nom_famille,
(SELECT b.nom FROM prd_activite b
 WHERE a.prd_activite_id = b.id LIMIT 1) as nom_activite,
c.codeproduit as code_produit
FROM `prd_famille` a
INNER JOIN prd_produit c
ON a.id = c.prd_famille_id
ORDER BY a.nom, c.codeproduit
;

```

Voici un petit échantillon du résultat obtenu :

nom_famille	nom_activite	code_produit
Aspirateurs électriques	Motoculture	SE 61
Aspirateurs électriques	Motoculture	SE 61 E
Broyeurs	Motoculture	GB 370
Broyeurs	Motoculture	GB 370 S
Broyeurs	Motoculture	GE 103
Broyeurs	Motoculture	GE 105
Broyeurs	Motoculture	GE 150
Broyeurs	Motoculture	GE 250
Broyeurs	Motoculture	GE 35 L
Débroussailleuses	Motoculture	FS 100
Débroussailleuses	Motoculture	FS 100 R
Débroussailleuses	Motoculture	FS 130

Et si je veux afficher les tarifs de chaque produit, je peux faire pareil avec une jointure entre la table des produits et la table des prix de vente ?

Un ange passe... suivi d'un petit sourire énigmatique du prof (certains élèves diraient « petit sourire sadique du prof »). Les élèves commencent à se trémousser sur leurs chaises, en se disant qu'il doit y avoir un piège... car il y en a un, forcément !! 😊

Bon, pour expliquer le fond du problème, je vous propose de partir d'une requête toute simple liant uniquement les tables produit et prix de vente (les libellés famille et activité, laissons-les de côté, on les rajoutera plus tard).

Commençons par une première jointure :

```

SELECT a.codeproduit as code_produit,
a.id as produit_id,
b.prd_produit_id, b.date_deb, b.prix
FROM prd_produit a
LEFT JOIN prd_prixstd b
ON a.id = b.prd_produit_id
ORDER BY a.codeproduit, b.date_deb
;

```


Un petit échantillon du résultat produit :

code_produit	produit_id	prd_produit_id	date_deb	prix
BR 500	80	80	2017-01-01	739.97000
BR 550	81	81	2017-01-01	756.69000
BR 600	82	82	2017-01-01	848.66000
BT 121 C	83	NULL	NULL	NULL
BT 45 Perceuse à bois	31	NULL	NULL	NULL
BT 45 Tarière	27	27	2017-01-01	581.10000
FR 130 T	121	NULL	NULL	NULL
FR 350	47	NULL	NULL	NULL
FR 450	48	NULL	NULL	NULL
FS 100	33	NULL	NULL	NULL

Intéressant non ? On voit que certains produits n'ont pas de tarif. Bon, vu la taille du jeu de données relatif aux prix de vente, il fallait s'y attendre.

Pour la suite de la démonstration, je vous propose d'éliminer les produits n'ayant pas de tarif, du coup remplacez « LEFT JOIN » par « INNER JOIN » et relancez la requête :

code_produit	produit_id	prd_produit_id	date_deb	prix
BR 500	80	80	2017-01-01	739.97000
BR 550	81	81	2017-01-01	756.69000
BR 600	82	82	2017-01-01	848.66000
BT 45 Tarière	27	27	2017-01-01	581.10000
GB 370 S	36	36	2017-01-01	1040.97000
GE 103	11	11	2017-01-01	268.39000
GE 105	26	26	2017-01-01	325.25000
GE 150	55	55	2017-01-01	394.65000
GE 250	56	56	2017-01-01	517.56000

OK, c'est mieux comme ça.

Maintenant, si vous observez le résultat obtenu, vous constatez que tous les prix de vente sont définis à la même date, soit le 1^{er} janvier 2017. Que se passerait-il si on avait un autre prix de vente au 1^{er} février 2017 ? Pour le savoir, je vous propose d'injecter artificiellement un jeu de données dans la table des prix de vente. Pour faire cela facilement, nous pouvons écrire une requête qui va réaliser les actions suivantes :

- sélectionner les prix de vente existants,
- appliquer à chaque prix de vente une augmentation de 10%
- injecter les nouveaux prix avec comme date d'application le 1^{er} février 2017

Une première requête pour préparer le terrain :

```
1 SELECT prd_produit_id,
2     '2017-02-01' as new_date_deb,
3     prix as ancien_prix,
4     prix*1.1 as nouveau_prix
5 FROM prd_prixstd
6 WHERE date_deb = '2017-01-01';|
```


Petit échantillon du résultat obtenu :

prd_produit_id	new_date_deb	ancien_prix	nouveau_prix
60	2017-02-01	459.03000	504.933000
61	2017-02-01	500.84000	550.924000
62	2017-02-01	584.45000	642.895000
51	2017-02-01	1109.00000	1219.900000
52	2017-02-01	1325.00000	1457.500000
53	2017-02-01	1620.00000	1782.000000
75	2017-02-01	2390.00000	2629.000000
76	2017-02-01	2910.00000	3201.000000
77	2017-02-01	3415.00000	3756.500000
126	2017-02-01	1019.23000	1121.153000
124	2017-02-01	868.73000	955.603000
125	2017-02-01	952.24000	1047.574000

Ok, c'est bon... Allez hop, je réutilise ma requête SELECT (en retirant quand même la colonne « ancien prix »), et j'injecte le résultat obtenu dans la table des prix de vente :

```
INSERT INTO prd_prixstd (prd_produit_id, date_deb, prix)
SELECT prd_produit_id, '2017-02-01', prix * 1.1
FROM prd_prixstd
WHERE date_deb = '2017-01-01';
```

PHPMyAdmin a l'air plutôt content, c'est bon signe 😊

 57 lignes insérées.
 Identifiant de la ligne insérée : 171 (traité en 0,0000 seconde(s))

```
INSERT INTO prd_prixstd (prd_produit_id, date_deb, prix) SELECT prd_produit_id, '2017-02-01', prix * 1.1 FROM prd_prixstd WHERE date_deb = '2017-01-01'
```

Relançons notre requête d'affichage des produits avec leurs prix :

```
SELECT a.codeproduit as code_produit,
       a.id as produit_id,
       b.prdr_produit_id, b.date_deb, b.prix
FROM prdr_produit a
INNER JOIN prdr_prixstd b
  ON a.id = b.prdr_produit_id
ORDER BY a.codeproduit, b.date_deb
;
```

Oups ! Y'aurait pas un petit problème ?

code_produit	produit_id	prdr_produit_id	date_deb	prix
BR 500	80	80	2017-01-01	739.97000
BR 500	80	80	2017-02-01	813.96700
BR 550	81	81	2017-01-01	756.69000
BR 550	81	81	2017-02-01	832.35900
BR 600	82	82	2017-01-01	848.66000
BR 600	82	82	2017-02-01	933.52600
BT 45 Tarière	27	27	2017-01-01	581.10000
BT 45 Tarière	27	27	2017-02-01	639.21000
GB 370 S	36	36	2017-01-01	1040.97000
GB 370 S	36	36	2017-02-01	1145.06700
GE 103	11	11	2017-01-01	268.39000
GE 103	11	11	2017-02-01	295.22900

Je comptais obtenir une ligne par produit, avec pour chaque produit son tarif, mais là je me retrouve avec deux lignes par produit.

J'en vois déjà qui vont me rétorquer...

« on n'a qu'à considérer que le tarif en vigueur, c'est celui du 1^{er} février. Du coup on met une clause WHERE avec sélection des tarifs au 1^{er} février, et le tour est joué !! »

C'est bien essayé, mais je ne suis pas d'accord. Certes, notre augmentation de février est une augmentation en masse, qui a été appliquée de manière uniforme sur les produits déjà tarifés du catalogue. Mais il s'agissait là d'un cas d'école. Il est rare que, dans la « vraie vie », on fasse des augmentations de ce type. Dans la « vraie vie », on applique des modifications de tarif au cas par cas, quelquefois pour une seule activité, quelquefois pour une ou plusieurs familles de produits, quelquefois produit par produit. Et tout cela à des dates très souvent différentes. C'est quoi dans ce cas, le « tarif en vigueur » ?

Pour essayer de cerner le problème, je vous propose d'appliquer quelques augmentations sur les produits ayant les identifiants 80, 81 et 82. Mais pour se rapprocher de la « vraie vie », je ne vais pas appliquer ces augmentations de manière uniforme : quelquefois j'augmenterai un seul produit, quelquefois deux, quelquefois les 3. Je vais étaler ces augmentations du 1^{er} mars au 1^{er} juin 2017 :

```
INSERT INTO prd_prixstd (prd_produit_id, date_deb, prix)
SELECT prd_produit_id, '2017-03-01', prix * 1.05
FROM prd_prixstd
WHERE date_deb = '2017-02-01'
and prd_produit_id in (80, 81, 82)
;
```

```
INSERT INTO prd_prixstd (prd_produit_id, date_deb, prix)
SELECT prd_produit_id, '2017-03-15', prix * 1.02
FROM prd_prixstd
WHERE date_deb = '2017-03-01'
and prd_produit_id in (80, 82)
;
```

```
INSERT INTO prd_prixstd (prd_produit_id, date_deb, prix)
SELECT prd_produit_id, '2017-04-01', prix * 1.02
FROM prd_prixstd
WHERE date_deb = '2017-03-15'
and prd_produit_id in (81, 82)
;
```

```
INSERT INTO prd_prixstd (prd_produit_id, date_deb, prix)
SELECT prd_produit_id, '2017-05-01', prix * 1.02
FROM prd_prixstd
WHERE date_deb = '2017-04-01'
and prd_produit_id in (82)
;
```

```
INSERT INTO prd_prixstd (prd_produit_id, date_deb, prix)
SELECT prd_produit_id, '2017-06-01', prix * 1.1
FROM prd_prixstd
WHERE date_deb = '2017-05-01'
and prd_produit_id in (80, 81, 82)
;
```

```
INSERT INTO prd_prixstd (prd_produit_id, date_deb, prix)
SELECT prd_produit_id, '2017-06-01', prix * 1.1
FROM prd_prixstd
WHERE date_deb = '2017-03-01'
and prd_produit_id in (80, 81)
```

OK, relançons maintenant la requête de consultation en nous focalisant uniquement sur nos 3 produits :

```
SELECT a.codeproduit as code_produit,
       a.id as produit_id,
       b.prp_produit_id, b.date_deb, b.prix
FROM prd_produit a
INNER JOIN prd_prixstd b
  ON a.id = b.prp_produit_id
WHERE prd_produit_id in (80, 81, 82)
ORDER BY a.codeproduit, b.date_deb
;
```

Cette fois ci il ne s'agit plus d'un échantillon, mais du jeu de données complet correspondant aux 3 produits BR500, BR550 et BR600 :

code_produit	produit_id	prp_produit_id	date_deb	prix
BR 500	80	80	2017-01-01	739.97000
BR 500	80	80	2017-02-01	813.96700
BR 500	80	80	2017-03-01	854.66535
BR 500	80	80	2017-03-15	871.75866
BR 500	80	80	2017-06-01	940.13189
BR 550	81	81	2017-01-01	756.69000
BR 550	81	81	2017-02-01	832.35900
BR 550	81	81	2017-03-01	873.97695
BR 550	81	81	2017-06-01	961.37465
BR 600	82	82	2017-01-01	848.66000
BR 600	82	82	2017-02-01	933.52600
BR 600	82	82	2017-03-01	980.20230
BR 600	82	82	2017-03-15	999.80635
BR 600	82	82	2017-04-01	1019.80248
BR 600	82	82	2017-05-01	1040.19853
BR 600	82	82	2017-06-01	1144.21838

Supposons que je souhaite connaître les tarifs applicables pour ces 3 produits, en date du 20 mars 2017. Pour ce faire, j'ai besoin de ma requête initiale, mais j'ai besoin de la compléter d'une sous-requête scalaire, placée au niveau de la clause WHERE, et qui va avoir pour rôle de rechercher pour chaque produit, le tarif le plus proche de la date demandée.

Voici ce qu'il faut écrire, avec la sous-requête scalaire « stabilotée » en jaune :

```
SELECT a.codeproduit as code_produit,
       a.id as produit_id,
       b.prp_produit_id, b.date_deb, b.prix
FROM prp_produit a
INNER JOIN prp_prixstd b
  ON a.id = b.prp_produit_id
  and b.date_deb = (
    SELECT MAX(c.date_deb) FROM prp_prixstd c
    WHERE c.prp_produit_id = b.prp_produit_id
    AND c.date_deb <= '2017-03-20'
  )
WHERE prp_produit_id in (80, 81, 82)
ORDER BY a.codeproduit, b.date_deb
;
```

Dans la sous-requête scalaire (stabilotée) de recherche du tarif le plus proche de la date d'application souhaitée (qui est ici le 20 mars 2017), vous noterez la présence de la fonction MAX(). Cette fonction est très importante, sans elle il est impossible de récupérer la date d'application la plus proche de la date souhaitée). L'autre élément important, c'est l'opérateur de comparaison « <= » (inférieur ou égal) qui permet de sélectionner la date antérieure la plus proche de celle recherchée (elle peut dans certains cas être égale à la date recherchée).

Notre requête est paramétrée pour une recherche au 20 mars 2017, alors testons-la :

code_produit	produit_id	prp_produit_id	date_deb	prix
BR 500	80	80	2017-03-15	871.75866
BR 550	81	81	2017-03-01	873.97695
BR 600	82	82	2017-03-15	999.80635

Nous récupérons deux prix en date du 15 mars, et un prix en date du 1^{er} mars (pour le produit BR550 qui n'a pas de prix au 15 mars). Cela semble bien fonctionner, alors poursuivons les tests.

Modifions la date (en rouge) dans la requête, en la paramétrant pour le 10 mars 2017 (soit avant l'augmentation du 15) ?

code_produit	produit_id	prp_produit_id	date_deb	prix
BR 500	80	80	2017-03-01	854.66535
BR 550	81	81	2017-03-01	873.97695
BR 600	82	82	2017-03-01	980.20230

Et au 10 juin 2017, ça donne quoi ?

code_produit	produit_id	prd_produit_id	date_deb	prix
BR 500	80	80	2017-06-01	940.13189
BR 550	81	81	2017-06-01	961.37465
BR 600	82	82	2017-06-01	1144.21838

Si vous prenez la peine de regarder de quelle manière certaines applications e-commerce gèrent les tarifs, vous vous apercevrez que ces applications utilisent des méthodes bien archaïques, si on les compare à ce que nous venons d'écrire. Certaines applications e-commerce ne savent tout simplement pas gérer de données tarifaires avec dates d'applications multiples, ou alors elles proposent un système tellement contraignant que les utilisateurs ne peuvent tout simplement pas s'en servir. Ces derniers se trouvent dès lors obligés de modifier les tarifs le jour J, éventuellement à minuit, pour application dès l'heure d'ouverture... Mais au fait, ça n'a pas d'heure d'ouverture un site e-commerce, c'est généralement ouvert en 7J/7 24H/24 !!

Quelquefois vous pourrez trouver des applications proposant une gestion de produits multi-dates, mais c'est le code de récupération des informations qui sera pourri. Car au lieu d'utiliser les techniques de jointure que nous venons d'étudier, les développeurs auront utilisé 2 requêtes distinctes traitées dans cet ordre :

- exécution d'une requête permettant de récupérer la liste des produits
- exécution d'une boucle PHP parcourant le résultat de la 1^{ère} requête et exécutant une nouvelle requête à chaque itération pour récupérer le tarif le plus proche

Dans ce que je viens de décrire, on peut trouver différentes manières de procéder, allant du pire au meilleur. Le pire étant le code SQL créé par concaténation sans utiliser la technique du SQL paramétré (d'un point de vue de la sécurité, c'est l'horreur).

Le seul cas qui pourrait justifier de ne pas utiliser la technique présentée dans ce chapitre, ce serait le cas où la table des produits se trouverait dans une base de données, et la table des tarifs se trouverait dans une autre base de données. Dans un cas comme celui là, impossible d'établir une jointure simple entre les tables « produit » et « prix de vente », il vaut mieux envisager l'utilisation de 2 requêtes distinctes.

Avant de poursuivre, je vous propose d'ajouter à notre requête précédente les libellés « famille » et « activité » de chaque produit :

```
SELECT a.codeproduit as code_produit,
       pf.nom as nom_famille,
       (SELECT b.nom FROM prd_activite b
        WHERE pf.prd_activite_id = b.id) as nom_activite,
       b.date_deb as date_tarif, b.prix as prix_vente
FROM prd_produit a
INNER JOIN prd_famille pf
  ON a.prd_famille_id = pf.id
INNER JOIN prd_prixstd b
  ON a.id = b.prd_produit_id
  and b.date_deb = (
```

```
        SELECT MAX(c.date_deb) FROM prd_prixstd c
        WHERE c.prd_produit_id = b.prd_produit_id
        AND c.date_deb <= '2017-03-20'
    )
WHERE a.prd_famille_id between 1 and 100
ORDER BY a.codeproduit, b.date_deb
;
```

Si vous avez trouvé ce chapitre ardu, vous avez bien raison. Vous avez le droit de « laisser décanter » tout ça et d'y revenir plus tard.

Le lecteur que ce sujet a intéressé pourra trouver un approfondissement sur le même thème dans l'article que j'ai publié dans GNU Linux Magazine de mars 2018 (n° 213) :

<https://boutique.ed-diamond.com/anciens-numeros/1305-gnulinix-magazine-213.html>

3.9 Les autres objets SQL

3.9.1 Les indexs

Nous allons aborder dans ce chapitre l'étude d'un objet SQL un peu particulier : l'index SQL.

Je dis qu'il est particulier, car il s'agit d'un objet purement technique, destiné à améliorer les performances des requêtes SQL.

Concrètement, la création d'un index se fait de cette façon :

```
CREATE INDEX index_name  
ON table_name (column1, column2, ...);
```

On donne un nom à l'index, on indique ensuite à quelle table il fait référence, et enfin on indique entre parenthèses la (ou les) colonne(s) de la table concernée par l'index. Cet index va contenir, après création, un jeu d'informations lui permettant d'accéder plus rapidement aux données, en tenant compte de l'ordre des colonnes indiquées entre parenthèses.

L'index ainsi créé est réactualisé en permanence, en fonction des insertions (INSERT), mises à jour (UPDATE) et suppressions (DELETE) qui sont effectuées sur la table sous-jacente.

Attention : l'index ne contient pas une copie des données de la table sous-jacente, il contient uniquement le « chemin d'accès » aux données de la table sous-jacente... On pourrait parler d'itinéraire, si le terme vous « parle » davantage, ou pour imaginer un peu plus, on pourrait que l'index est une sorte de GPS permettant d'accéder plus rapidement aux données qui nous intéressent. Quelle que soit l'image qui vous convient, retenir que le terme « chemin d'accès » (ou en anglais « data path ») est celui qui est le plus couramment utilisé dans le jargon SQL.

Supposons que vous lanciez plusieurs fois par jour, une requête SQL dont le rôle consiste à extraire des données en provenance de tables contenant plusieurs dizaines de milliers de lignes, voire beaucoup plus...

Si le moteur SQL ne dispose d'aucun index sur la (ou les) table(s) concernée(s) par la requête, alors il va être obligé de parcourir la totalité des lignes de la table (ou des tables s'il y a jointure) pour identifier et extraire les lignes concernées par la requête. Enfin ce que je dis est inexact, ou plutôt cela dépend beaucoup du SGBD et des caractéristiques de l'analyseur de données que le SGBD utilise. Et cela varie aussi, pour un même moteur, en fonction des versions de ce moteur.

Donc, le moteur SQL peut utiliser plusieurs stratégies pour accéder aux données qui vous intéressent, les plus courantes étant :

- le parcours de l'intégralité d'une table (on dit que la table est « parsée » dans son intégralité),
- ou la création d'un index temporaire par le moteur.

Ce choix opéré par le moteur du SGBD dépendra notamment des statistiques que le moteur entretient en interne par rapport aux données contenues dans les tables.

En tant que développeurs, ou administrateurs de bases de données, nous pouvons essayer d'alléger le moteur SQL de son fardeau en pré-construisant des indexs SQL (via l'instruction CREATE INDEX que nous venons de voir). En fait, on essaie au travers de la création d'indexs d'influer sur la stratégie employée par le moteur SQL pour accéder aux données.

Une fois cet index créé, et si le SGBD estime qu'il est adapté à la requête SQL qu'on lui demande d'exécuter, ce dernier va utiliser cet index, s'épargnant ainsi le travail de construction de l'index, qui peut être chronophage, surtout si la table à indexer est de grande taille. Ah oui, j'ai oublié de vous dire que le CREATE INDEX peut être long, si la table concernée est volumineuse, il faut parfois s'armer de patience quand on crée des indexs sur des tables...

A l'usage, le développeur SQL n'utilisera jamais un index dans ses requêtes. Si l'on se base sur notre exemple de CREATE INDEX vu au début de ce chapitre, le développeur n'aura par exemple pas le droit d'écrire une requête de ce genre :

```
SELECT * FROM index_name WHERE ...
```

La requête ci-dessus ne fonctionnera pas. Le développeur SQL écrira ses requêtes en utilisant les tables (ou les vues) qui existent dans la base de données. C'est le moteur SQL qui fera le travail de recherche d'un index le plus adapté à la requête à traiter. S'il trouve un index adapté, il va s'en servir pour trouver plus rapidement les données à récupérer. S'il n'en trouve pas, il fera son propre travail de recherche des données, comme nous l'avons évoqué précédemment (avec peut être un parsing complet de la table, ou peut être la création d'un index temporaire).

Vous pourriez vous dire : « OK, c'est cool, je vais créer plein d'indexs sur mes tables, comme ça j'aurai des performances optimales ». PERDU !!! Car les indexs ont aussi un impact sur les performances, mais à un autre niveau. Comme je l'écrivais tout à l'heure, chaque fois que vous modifiez, ajoutez ou supprimez des données dans une table, cela déclenche la mise à jour des indexs liés à la table, et cette mise à jour va ralentir les INSERT, UPDATE et DELETE. Ce que vous gagnez d'un côté, vous le perdez de l'autre, donc soyez prudents et ne créez pas plus d'indexs que nécessaire.

A noter que l'on peut créer des indexs avec clé unique ce qui est très pratique lorsque l'on souhaite empêcher l'insertion de doublons dans une table. La création d'un index avec clé unique se fait de la façon suivante :

```
CREATE UNIQUE INDEX index_name  
ON table_name (column1, column2, ...);
```

Ce chapitre est une présentation succincte, mais néanmoins suffisante pour une première initiation.

Il existe quelques petites différences de syntaxe d'un SGBD à l'autre, que ce soit pour le CREATE INDEX comme pour le DROP INDEX (qui permet de supprimer un index). Si vous êtes amenés à travailler sur plusieurs SGBD, vous trouverez sur les pages suivantes quelques exemples adaptés à différents SGBD :

https://www.w3schools.com/sql/sql_create_index.asp

<https://www.w3resource.com/sql/creating-index/sql-creating-index.php>

Le site Openclassrooms propose un tuto présentant le fonctionnement des index de manière détaillée et précise. Je vous en recommande la lecture :

<https://openclassrooms.com/courses/administrez-vos-bases-de-donnees-avec-mysql/index-1>

3.9.2 Les vues

Les vues sont des objets SQL très intéressants à exploiter.

Prenons pour exemple l'un des dernières requêtes étudiées dans le chapitre 3.8.3, par exemple celle-ci :

```
SELECT etudiant, 'Java' as matiere, Java as note FROM etudnotes
UNION
SELECT etudiant, 'BD' as matiere, BD FROM etudnotes
UNION
SELECT etudiant, 'Algèbre' as matiere, Algebre FROM etudnotes
```

Supposons que vous travailliez pour un organisme de formation, et que vous deviez relancer plusieurs fois par semaine cette requête, pour réactualiser des données transmises à un autre service. Je vous parie que vous allez en avoir vite ras le bol, de lancer cette requête (que vous allez devoir copier-coller plusieurs fois par semaine dans votre client SQL préféré).

Vous pouvez bien évidemment intégrer cette requête SQL dans un script PHP, comme étudié dans le cours PHP (cf. github.com/gregja/PHPCorner), pourquoi pas ?

Mais supposons que des collègues vous demandent le code source de cette requête, pour pouvoir la lancer par eux-même quand ils en ont besoin. Vous allez bien évidemment leur transmettre le code, mais rien ne garantit qu'ils vont le réutiliser tel quel.

Pour couper court à tous ces problèmes du quotidien, je vous propose de créer un objet SQL particulier qu'on appelle une vue.

Si on reprend notre requête ci-dessus, voici le code permettant de créer une vue s'appuyant sur cette requête :

```
CREATE VIEW etud_mat AS
  SELECT etudiant, 'Java' as matiere, Java as note FROM etudnotes
  UNION
  SELECT etudiant, 'BD' as matiere, BD FROM etudnotes
  UNION
  SELECT etudiant, 'Algèbre' as matiere, Algebre FROM etudnotes ;
```

Voilà, vous venez de créer une vue qui s'appelle "etud_mat" et vous pouvez maintenant l'utiliser comme une simple table, via des requêtes de ce type :

```
SELECT * FROM etud_mat;
```

... ou encore :

```
SELECT * FROM etud_mat
WHERE matiere = 'BD'
ORDER BY etudiant, matiere;
```

Vous pouvez maintenant communiquer à vos collègues le nom de la vue (etud_mat) et ils vont pouvoir l'utiliser dans leurs propres requêtes. La complexité du code est masquée par la vue, et vos collègues ne risquent pas de

modifier ce code par erreur. Vous pouvez bien évidemment utiliser cette vue dans les requêtes intégrées dans vos scripts PHP si vous le souhaitez.

Les vues sont idéales pour embarquer du code “métier”, qui est très souvent complexe. N’hésitez pas à les utiliser chaque fois que vous développez une requête SQL complexe qui est susceptible d’être réutilisée régulièrement.

Une petite astuce à noter pour conclure ce chapitre : si vous avez perdu le code source ayant servi à créer une vue, vous pouvez facilement régénérer ce code en passant par la requête suivante :

```
SHOW CREATE VIEW etud_mat;
```

Voici le résultat obtenu :

View	Create View	character_set_client	collation_connection
etud_mat	CREATE ALGORITHM=UNDEFINED DEFINER=`root`@`localhost` SQL SECURITY DEFINER VIEW `etud_mat` AS select `etudnotes`.`ETUDIANT` AS `etudiant`,`Java` AS `matiere`,`etudnotes`.`JAVA` AS `note` from `etudnotes` union select `etudnotes`.`ETUDIANT` AS `etudiant`,`BD` AS `matiere`,`etudnotes`.`BD` AS `BD` from `etudnotes` union select `etudnotes`.`ETUDIANT` AS `etudiant`,`Algèbre` AS `matiere`,`etudnotes`.`ALGEBRE` AS `Algebre` from `etudnotes`	utf8mb4	utf8mb4_unicode_ci

Le code contenu dans la seconde colonne (Create View) est trop long pour s’afficher correctement dans le tableau ci-dessus, alors je l’ai extrait et l’ai copié-collé ci-dessous :

```
CREATE ALGORITHM=UNDEFINED DEFINER=`root`@`localhost` SQL SECURITY DEFINER VIEW `etud_mat` AS select
`etudnotes`.`ETUDIANT` AS `etudiant`,`Java` AS `matiere`,`etudnotes`.`JAVA` AS `note` from `etudnotes` union select
`etudnotes`.`ETUDIANT` AS `etudiant`,`BD` AS `matiere`,`etudnotes`.`BD` AS `BD` from `etudnotes` union select
`etudnotes`.`ETUDIANT` AS `etudiant`,`Algèbre` AS `matiere`,`etudnotes`.`ALGEBRE` AS `Algebre` from `etudnotes`
```

J’ai mis en marron la partie de la requête que vous pouvez réutiliser pour recréer la vue en cas de besoin. Bon la lisibilité du code en a pris un coup, certes, mais le code d’origine est bien là (légèrement transformé car toutes les colonnes sont préfixées avec le nom de leurs tables respectives). Vous noterez que MySQL a introduit des apostrophes inverse un peu partout, n’en tenez pas compte, ils n’ont pas d’importance.

Il est important de souligner que, chaque fois que vous exécutez une requête sur une vue, vous réexécutez le code SQL embarqué dans cette vue. Les données produites par cette vue ne sont pas stockées dans la vue, contrairement aux tables qui sont de véritables conteneurs de données. Donc n’espérez pas obtenir de meilleures performances en exécutant une requête SELECT sur une vue, par rapport à l’exécution manuelle du code source embarqué dans cette vue.

Certaines bases de données, comme par exemple DB2, fournissent un type d’objet particulier qu’on appelle une MQT (Materialized Query Table). Ce type d’objet, qui n’existe pas pour l’instant dans MariaDB, permet de disposer d’un objet hybride, qui est à la fois une vue et une table. Je ne rentre pas dans le détail ici, car le sujet n’est pas d’actualité sur MariaDB, mais comme les technos open-source évoluent très vite, on disposera peut être de ce type d’objet sur MariaDB, dans un avenir proche.

3.9.3 Les fonctions utilisateurs

Vous avez vu dans les chapitres précédents quelques exemples d'utilisation de fonctions SQL prédéfinies, comme par exemple la fonction UPPER() qui permet de mettre du texte en majuscule, ou la fonction LOWER() qui fait strictement l'inverse.

Vous serez forcément confronté à des cas de figure pour lesquels une fonction SQL spécifique vous serait utile, car elle vous permettrait de simplifier votre code. Malheureusement, la fonction en question n'existe pas dans MariaDB.

Heureusement SQL vous permet de créer vos propres fonctions, qu'on appelle « fonctions utilisateurs » (ou en anglais UDF pour « User Defined Functions »). Nous allons voir tout de suite un exemple.

Vous vous souvenez peut être que les noms de nos employés sont tous saisis en majuscule. Ce serait plus sympa si nous pouvions afficher le nom des employés en minuscule, avec le premier caractère en majuscule. Si ça se trouve, c'est une fonction qui existe déjà sur MariaDB, mais je m'en moque, je vais écrire ma propre version, et je vais l'appeler « capitalize ».

Voici le code de création de la fonction :

```
CREATE FUNCTION capitalize (param1 VARCHAR(255))  
RETURNS VARCHAR(255) DETERMINISTIC  
RETURN CONCAT(UPPER(LEFT(param1, 1)), MID(LOWER(param1), 2))
```

La première ligne définit le nom de la fonction (capitalize), ainsi que le nom, le type et la longueur maximale du paramètre reçu, qui sera utilisé à l'intérieur de la fonction comme une simple variable. Ici nous avons une variable « param1 » qui est de type VARCHAR et a une longueur maximale de 255 caractères. Vous noterez que nous pourrions avoir plusieurs paramètres en entrée, il suffirait de les séparer par une virgule.

La seconde ligne définit le type et la longueur maximale de la donnée renvoyée par la fonction. Ici nous avons un type VARCHAR et une longueur maximale de 255, soit le même type que le paramètre en entrée. C'est un cas particulier, nous pourrions avoir un type et une longueur différents en sortie, dans d'autres cas de figure. Vous noterez que la fonction ne renvoie qu'une seule valeur et c'est un point essentiel à prendre en considération : les fonctions ne sont pas capables de renvoyer plus d'une valeur à la fois.

Le mot clé DETERMINISTIC sur la seconde ligne a pour effet d'activer un cache géré par MariaDB, cache qui est associé à la fonction concernée. Si la fonction est à l'état DETERMINISTIC et qu'elle reçoit plusieurs fois la même valeur en entrée, elle n'exécutera le code sous-jacent que pour le premier appel, conservera le résultat dans le cache, et réutilisera le contenu du cache pour qu'un second appel sera réalisé avec un paramètre déjà employé. L'inverse de DETERMINISTIC, c'est NOT DETERMINISTIC (qui correspond au mode par défaut). Vous retrouverez cette notion (DETERMINISTIC) sur d'autres SGBD, elle n'est pas spécifique à MariaDB.

La troisième ligne contient, à droite du mot clé RETURN, le code « métier », celui qui va être exécuté à chaque appel de la fonction. Ici nous avons une concaténation de 2 valeurs :

- la première correspondant au premier caractère du paramètre en entrée mis en majuscule grâce à la fonction UPPER()
- la seconde correspondant à la suite du paramètre en entrée (à partir du second caractère), mis en minuscule grâce à la fonction LOWER()

Voici un exemple d'utilisation de notre fonction maison capitalize() :

```
SELECT empno, lastname, capitalize(lastname) as lastname_cap
FROM employes
```

empno	lastname	lastname_cap
20	THOMPSON	Thompson
60	STERN	Stern
100	SPENSER	Spenser
170	YOSHIMURA	Yoshimura
180	SCOUTTEN	Scoutten
190	WALKER	Walker
250	SMITH	Smith
280	SCHNEIDER	Schneider
300	SMITH	Smith
310	SETRIGHT	Setright

On peut améliorer la robustesse de notre fonction, en faisant en sorte qu'elle vérifie que le type de données entrante n'est pas « null ». Supprimons la première version :

```
DROP FUNCTION capitalize ;
```

et recréons la avec cette nouvelle version :

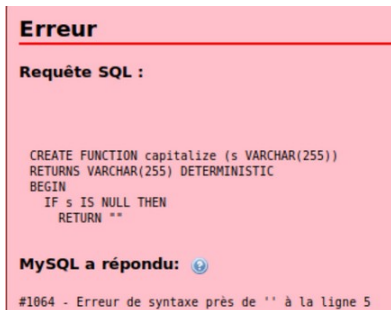
```
CREATE FUNCTION capitalize (param1 VARCHAR(255))
RETURNS VARCHAR(255) DETERMINISTIC
BEGIN
    IF param1 IS NULL THEN
        RETURN "";
    ELSE
        RETURN CONCAT(UPPER(LEFT(param1,1)),MID(LOWER(param1),2));
    END IF;
END;
```

Attention, vous allez avoir un petit problème lors de la compilation du code de cette fonction, mais j'en parlerai page suivante.

Pour l'instant concentrons-nous sur l'aspect fonctionnel de ce code. Dans cette nouvelle version, vous voyez que notre code est plus complexe. La présence des mots clés BEGIN et END nous permet en effet d'introduire plusieurs lignes de code SQL, séparées par des points virgules. Et nous sommes du coup en

mesure d'utiliser une structure telle que « IF ... THEN... ELSE », que vous connaissez certainement pour l'avoir utilisée dans d'autres langages (comme Javascript et PHP).

Si vous essayez d'exécuter ce code dans un client SQL, vous allez obtenir un message d'erreur comme celui ci-dessous (obtenu avec Phpmyadmin) :



Ce message d'erreur est bien étrange, d'autant que le code SQL est parfaitement correct. C'est dû au fait que l'interpréteur SQL de MariaDB (et de MySQL) interprète mal la présence des mots clé BEGIN et END, on peut dire qu'il est un peu « à la ramasse ». Heureusement, il existe une solution qui consiste à introduire des délimiteurs dans le code, comme dans l'exemple ci-dessous :

```
DELIMITER //
CREATE FUNCTION capitalize (param1 VARCHAR(255))
RETURNS VARCHAR(255) DETERMINISTIC
BEGIN
    IF param1 IS NULL THEN
        RETURN "";
    ELSE
        RETURN CONCAT(UPPER(LEFT(param1,1)),MID(LOWER(param1),2));
    END IF;
END;
//
DELIMITER;
```

Il faut reconnaître que c'est assez bizarre, mais ça fonctionne. Cela a pour effet d'échapper les mots clés BEGIN et END, du coup la compilation du code source de la fonction passe sans problème.

Cette technique propre à MariaDB et à MySQL fait penser un peu à la syntaxe HEREDOC du langage PHP : <http://php.net/manual/fr/language.types.string.php#language.types.string.syntax.heredoc>

Nous réutiliserons cette technique d'échappement dans le chapitre suivant, car les mots clés BEGIN et END sont naturellement présents dans ce type d'objet SQL.

3.9.4 Les procédures stockées

Les procédures stockées (en anglais : « stored procedures ») fonctionnent un peu sur le même principe que les fonctions, mais elles offrent beaucoup plus de liberté. On peut encapsuler plusieurs requêtes SQL indépendantes au sein d'une procédure, y intégrer des requêtes de mise à jour, de suppression et d'insertion, et même des instructions de création d'objet comme CREATE TABLE... En fait on peut écrire de véritables programmes grâce aux procédures stockées. C'est très puissant, et le suprême avantage, c'est que le code est embarqué au plus près des données, et que vous obtiendrez des performances optimales.

Comme pour les fonctions du chapitre précédent, la compilation de procédures stockées nécessite l'emploi des délimiteurs.

Voici un exemple de procédure stockée qui reçoit en entrée un code département, et renvoie en sortie le nombre d'employés de ce département, ainsi que le cumul des salaires de ce département :

```
DELIMITER //
CREATE PROCEDURE stats_salaires(IN p_dept INT,
                                OUT p_comptage INT,
                                OUT p_total_sal FLOAT)
NOT DETERMINISTIC
LANGUAGE SQL
BEGIN
    # comptage des employés et somme des salaires
    SELECT count(*), sum(salaires.montant)
        INTO p_comptage, p_total_sal
    FROM employes
    INNER JOIN salaires
        ON employes.empno = salaires.empno
    WHERE employes.workdept = p_dept;
END;
//
DELIMITER;
```

Pour tester cette procédure stockée, saisissez les 2 lignes de code suivantes dans votre client SQL préféré :

```
CALL stats_salaires(1,@compteur,@tot_sal);
```

```
SELECT @compteur, @tot_sal;
```

```
+-----+-----+
| @compteur | @tot_sal |
+-----+-----+
| 1         | 45000    |
+-----+-----+
```

Parmi les points importants à souligner dans cette procédure, il y a cette structure SELECT particulière qu'on appelle « SELECT INTO » :

```
SELECT count(*), sum(salaires.montant)
INTO p_comptage, p_total_sal
FROM ...
```

Grâce à cette structure, nous sommes en mesure d'alimenter directement des variables internes à la procédure. Dans notre cas, les variables alimentées sont les paramètres de sortie de la procédure. Mais on peut aussi créer des variables internes, qui ne sont pas des paramètres d'entrée/sortie. Pour le vérifier, je vous invite à supprimer la procédure précédente en fait un « DROP PROCEDURE stats_salaires », puis à la recréer avec la variante ci-dessous :

```
DELIMITER //
CREATE PROCEDURE stats_salaires(IN p_dept INT,
                                OUT p_comptage INT,
                                OUT p_total_sal FLOAT)
NOT DETERMINISTIC
LANGUAGE SQL
BEGIN
    DECLARE tmp_comptage INT DEFAULT 0;
    DECLARE tmp_total_sal FLOAT DEFAULT 0;

    # comptage des employés et somme des salaires
    SELECT count(*), sum(salaires.montant)
        INTO tmp_comptage, tmp_total_sal
    FROM employes
    INNER JOIN salaires
        ON employes.empno = salaires.empno
    WHERE employes.workdept = p_dept;

    SET p_comptage = tmp_comptage;
    SET p_total_sal = tmp_total_sal;
END;
//
DELIMITER;
```

Dans le code précédent, j'ai fait ressortir les principaux changements en gras. Il faut souligner que toutes les variables internes doivent être déclarées au tout début de la procédure, juste après le BEGIN, et avant tout autre type d'instruction. C'est une contrainte que l'on retrouve sur d'autres SGBD, MariaDB n'est pas le seul à l'imposer.

Vous aurez sans doute remarqué la présence des mots clés « NOT DETERMINISTIC ». Leur présence est nécessaire pour informer le moteur SQL du fait que le code interne à la procédure doit être réexécuté à chaque appel, et ce quelles que soient les valeurs transmises en entrée de la procédure. Les données

produites par la procédure sont susceptibles de fluctuer, et on souhaite obtenir à chaque appel des données les plus « fraîches » possibles.

Pour vous montrer que les procédures stockées sont vraiment très souples et très puissantes, je vous propose de créer une variante de la procédure précédente. Voici le code, je le commente tout de suite après :

```
DELIMITER //
```

```
CREATE PROCEDURE stats_salaires2(IN p_dept INT,  
                                OUT p_comptage INT,  
                                OUT p_total_sal FLOAT)
```

```
NOT DETERMINISTIC  
LANGUAGE SQL  
BEGIN  
    DECLARE tmp_comptage INT DEFAULT 0;  
    DECLARE tmp_total_sal FLOAT DEFAULT 0;  
    # comptage des employés et somme des salaires  
  
    DROP TABLE IF EXISTS tmp_datas;  
  
    CREATE TEMPORARY TABLE tmp_datas  
    SELECT count(*) as comptage, sum(salaires.montant) as tot_sal  
    FROM employes  
    INNER JOIN salaires  
        ON employes.empno = salaires.empno  
    WHERE employes.workdept = p_dept;  
  
    SELECT comptage, tot_sal  
        INTO tmp_comptage, tmp_total_sal  
    FROM tmp_datas;  
  
    SET p_comptage = tmp_comptage;  
    SET p_total_sal = tmp_total_sal;  
END;  
//  
DELIMITER;
```

Dans la variante de procédure ci-dessus, j'ai introduit des requêtes supplémentaires, avec notamment la création d'une table temporaire :

```
CREATE TEMPORARY TABLE tmp_datas  
SELECT count(*) as comptage, sum(salaires.montant) as tot_sal  
FROM employes  
INNER JOIN salaires  
    ON employes.empno = salaires.empno  
WHERE employes.workdept = p_dept;
```

Nous n'avions pas encore vu cette technique, qui nous permet de disposer d'un jeu de données temporaire, disponible le temps de la session SQL. Cette table temporaire disparaîtra dès que notre

session SQL se terminera. Cette technique est aussi un moyen de pallier l'absence des CTE (Common Table Expression) dans les versions de MariaDB antérieures à la 10.2.

Vous noterez que, juste avant de créer la table temporaire, j'ai pris la peine d'ajouter la ligne suivante :

```
DROP TABLE IF EXISTS tmp_datas;
```

Il s'agit d'une précaution visant à s'assurer que la table temporaire n'est pas déjà présente dans l'environnement de données. Supposons que nous appelions la même procédure deux fois de suite, sans le DROP TABLE de cette table temporaire (uniquement si elle existe), le second appel déclencherait un plantage, car le CREATE TABLE TEMPORARY TABLE échouerait dans ce cas de figure.

Mais revenons à la table temporaire elle-même. Vous voyez que cette table temporaire est réutilisée dans la requête SQL suivante, avec la technique du SELECT INTO vue précédemment :

```
SELECT comptage, tot_sal  
  INTO tmp_comptage, tmp_total_sal  
 FROM tmp_datas;
```

Bon, cet exemple est un peu « capilotracté », compte tenu que la version précédente faisait la même chose avec beaucoup moins de code, mais je pense que cela vous donne une idée des nombreuses possibilités offertes par les procédures stockées.

Une petite astuce pour finir : dans notre dernier exemple, nous avons créé une table temporaire à l'intérieur d'une procédure stockée. Eh bien cette table temporaire est accessible à l'extérieur de la procédure, immédiatement après le CALL de cette dernière. Vous pouvez donc écrire ceci :

```
CALL stats_salaires(1,@compteur,@tot_sal);
```

```
select * from tmp_datas;
```

Cela pourra vous rendre service dans certains cas, de pouvoir ainsi analyser le contenu de certaines tables temporaires, pour vous assurer que leur contenu est correct.

Pour une présentation plus détaillée sur les procédures stockées, je recommande de commencer par la page suivante :

<https://mariadb.com/kb/en/library/create-procedure/>

Vous trouverez sur internet de nombreux tutoriaux dédiés aux procédures stockées, celui-ci me semble particulièrement complet :

<https://www.w3resource.com/mysql/mysql-procedure.php>

Une bonne introduction au sujet est également celle proposée par Openclassrooms dans ce tuto :

<https://openclassrooms.com/courses/administrez-vos-bases-de-donnees-avec-mysql/procedures-stockees>

4 Conclusion

Plutôt qu'une présentation très détaillée (que vous trouverez dans d'autres ouvrages cités en référence), j'ai préféré vous proposer un tour d'horizon de SQL, de manière à vous permettre de vous faire une idée plus précise sur l'énorme potentiel qu'offre ce langage de programmation. Car SQL est bien un véritable langage de programmation, puissant, riche, performant, et pas un simple outil de stockage de données, comme voudraient le faire croire les partisans de l'approche « NoSQL ».

J'ai profité de la rédaction de ce support pour mettre l'accent sur un certain nombre de cas pratiques qui pourront vous être utiles au quotidien (comme la suppression de doublons et d'orphelins).

Je suis passé complètement à côté d'un certain nombre de sujets, comme par exemple les contraintes d'intégrité référentielle, les triggers (déclencheurs), etc. Je développerai certainement ces sujets dans une prochaine version de ce support de cours. Autre sujet que j'ai passé sous silence ici, c'est l'ordre SQL ALTER, qui permet de modifier la structure d'une table en ajoutant ou supprimant des colonnes ou en modifiant leur type et leur longueur. C'est un sujet à part entière que je n'ai malheureusement pas le temps de développer dans cette version du cours (peut être pour une prochaine version).

Je vous encourage à étudier les ouvrages que j'ai référencés dans le chapitre suivant. Et je vous encourage également à étudier les chapitres relatifs à SQL qui se trouvent dans le support de cours PHP disponible ici :

<https://github.com/gregja/PHPCorner>

... vous trouverez dans ce support complémentaire un exemple d'application PHP s'appuyant sur une base SQL proche de ce que l'on peut trouver en entreprise, avec en complément quelques techniques de niveau avancé sur la manipulation de données temporelles.

Je suis également passé très vite sur la partie DDL (Data Definition Language) et je ne suis pas rentré dans les subtilités relatives aux clés primaires auto-incrémentées. Vous trouverez une meilleure introduction sur ce sujet dans le support de cours PHP.

Phpmyadmin est un formidable couteau suisse, avec ses nombreux assistants, mais j'ai préféré ne pas trop insister sur ses capacités, car j'ai préféré privilégier une approche « full SQL ». Je crois en effet plus utile que vous maîtrisiez la syntaxe SQL plutôt que les raccourcis proposés par Phpmyadmin, cela vous permettra de passer plus facilement sur d'autres SGBD par la suite.

5 Annexe

5.1 Bibliographie

SQL Antipatterns,

Avoiding the Pitfalls of Database Programming

by Bill Karwin

<https://pragprog.com/book/bksqla/sql-antipatterns>

=> un guide des mauvaises pratiques SQL, et qui explique comment les corriger (très intéressant)

SQL Hacks

Tips & Tools for Digging Into Your Data

By Andrew Cumming, Gordon Russell

Publisher: O'Reilly Media

=> traduit en français sous le titre "SQL à 200%" (épuisé, après la fermeture de O'Reilly France, très bon bouquin que vous pouvez peut être trouver d'occasion). L'ouvrage original en anglais est disponible sur oreilly.com et chez Amazon.

SQL Cookbook

Query Solutions and Techniques for Database Developers

By Anthony Molinaro

Publisher: O'Reilly Media

=> traduit en français sous le titre "SQL par l'exemple" (épuisé, après la fermeture de O'Reilly France, mais très bon bouquin que vous pouvez peut être trouver d'occasion, il est très complémentaire de SQL Hacks). L'ouvrage original en anglais est disponible sur oreilly.com et chez Amazon.

SQL avancé

Programmation et techniques avancées

par Joe Celko

Editions Vuibert (2000)

=> le seul livre en français de cette sélection, un ouvrage de référence

Seven Databases in Seven Weeks

A Guide to Modern Databases and the NoSQL Movement

by Eric Redmond and Jim R. Wilson

=> pour élargir son horizon à d'autres SGBD

MySQL, 5th Edition

By Paul Dubois

Published Apr 2, 2013 by Addison-Wesley Professional.

=> un ouvrage de référence par un spécialiste de MySQL (le livre ne traite pas de MariaDB, mais l'essentiel du contenu de l'ouvrage est compatible avec MariaDB)

<http://www.informit.com/store/mysql-9780321833877>

Dans le **n° 211 de GNU Linux Magazine (janvier 2018)**, j'ai publié un article présentant un certain nombre de nouveautés de MariaDB en version 10.2, dont les CTE (Common Table Expressions) et les fonctions de fenêtrage (Window Functions) :

<https://boutique.ed-diamond.com/en-kiosque/1287-gnulinix-magazine-211.html>

Dans le n° 213 de GNU Linux Magazine (mars 2018), j'ai publié un article consacré sur la gestion des dates en SQL. Par rapport au même sujet traité dans ce dossier, j'ai introduit des problématiques de chevauchement de date, qui rendent certaines requêtes plus complexes) :

<https://boutique.ed-diamond.com/anciens-numeros/1305-gnulinix-magazine-213.html>

Le hors série **n° 88 de GNU Linux Magazine** est consacré à MariaDB, et approfondit de nombreux domaines que je n'ai pas pu traiter dans le présent support :

https://boutique.ed-diamond.com/les-guides/1159-gnulinix-magazine-hs-88.html#/37-format_du_magazine-magazine_papier

Pour finir, voici une petite sélection de livres sur SQL en français :

<https://sgbd.developpez.com/livres/index/?page=Les-livres-en-francais#L2841774686>

On trouve aussi de bons bouquins consacrés à SQL sur Leanpub, souvent à des prix très abordables :

https://leanpub.com/bookstore/type/book/sort/earnings_in_last_7_days?search=sql

5.2 Liens utiles

Cours SQL en ligne :

<http://sql.sh/>

<https://www.w3schools.com/sql/>

<https://openclassrooms.com/courses/administrez-vos-bases-de-donnees-avec-mysql>

<https://openclassrooms.com/courses/administrez-vos-bases-de-donnees-avec-mysql/procedures-stockees>

Assistant de formatage de code SQL (très pratique pour remettre d'aplomb un code SQL confus) :

http://www.dpriver.com/pp/sqlformat.htm?ref=g_wangz

Générateur de jeux de données SQL :

<http://mockaroo.com/>

Pourquoi il est préférable d'écrire et de maîtriser son code SQL, plutôt que de dépendre d'un ORM (Object Relational Mapping). Un point de vue auquel j'adhère complètement :

<https://medium.com/bumpers/our-go-is-fine-but-our-sql-is-great-b4857950a243>

5.3 Problème de connexion MariaDB avec Netbeans

Dans le chapitre 2.3.1, nous avons vu comment configurer Netbeans pour l'utiliser en tant que client SQL avec MariaDB.

Mais si vous êtes sous certaines versions de Linux (comme par exemple la distribution Mageia) ou sous Mac OSX, alors vous avez probablement rencontré le message d'erreur suivant :

```
Cannot establish a connection to jdbc:mysql://localhost:3306/mysql?  
zeroDateTimeBehavior=convertToNull using com.mysql.jdbc.Driver
```

Le problème provient de la configuration du firewall.

Je vais indiquer ci-dessous comment corriger le problème sous Mageia, si vous êtes sur Mac OSX, vous devrez trouver les équivalences permettant d'appliquer les mêmes réglages.

Sur Mageia, passer en ligne de commande en mode administrateur, puis ouvrez avec votre éditeur préféré le fichier suivant (par exemple avec vim ou avec kwrite) :
`/etc/my.cnf`

Appliquez les modifications suivantes :

- mettre en commentaire la ligne suivante :
`#skip-networking`

- modifier l'adresse IP du paramètre bind-address de la façon suivante :
`bind-address=0.0.0.0`

Sauvegardez le fichier modifié.

Allez dans la partie « Sécurité » du centre de contrôle de Mageia :



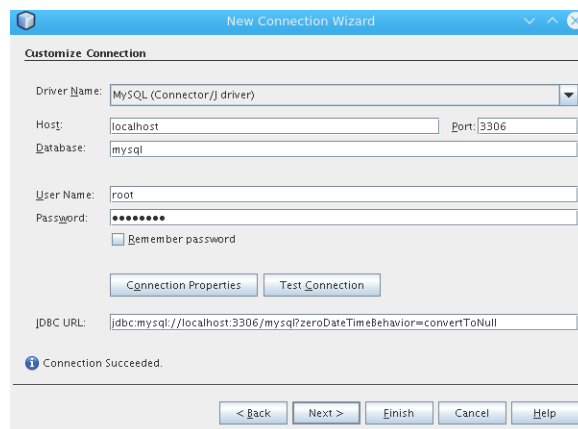
Cliquez sur « Configurer votre pare-feu personnel » et cocher l'option « Serveur MySQL » :



Puis cliquez sur « OK » plusieurs fois.

Relancez le serveur MySQL :
service mysqld restart

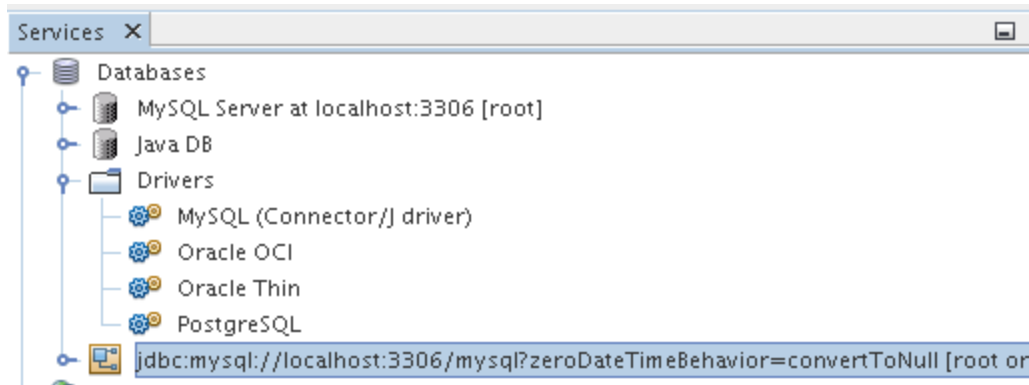
Puis retournez sur Netbeans et recliquez sur « Test connection » :



« Connexion Succeeded » ? C'est parfait, cliquez sur « Finish ».

Vous voyez maintenant apparaître une nouvelle ligne qui correspond à un connecteur MySQL

prêt à l'emploi.



Vous pouvez faire un clic-droit sur cette nouvelle ligne et sélectionner l'option « Execute Command ».

Quelques liens utiles :

- config d'un pare-feu sur Linux Mageia

<https://doc.mageia.org/mcc/3/fr/content/drakfirewall.html>

- config d'un environnement LAMP sur Linux Mageia :

https://www.linuxtricks.fr/wiki/wiki.php?id_contents=153

<http://alcatiz.developpez.com/tutoriel/installer-lamp-mageia/>

5.4 Problème de configuration avec Phpmyadmin

Vous rencontrerez peut être des problèmes pour faire fonctionner Phpmyadmin, cela m'est arrivé sur un environnement Linux Ubuntu.

La page décrivant l'installation de Phpmyadmin en environnement Ubuntu est la suivante.

<https://doc.ubuntu-fr.org/phpmyadmin>

Si en allant sur la page suivante, le serveur Apache vous indique que cette page est inexistante :

<http://localhost/phpmyadmin>

... alors commencez par essayer ceci en ligne de commande :

```
sudo ln -s /usr/share/phpmyadmin /var/www
```

Cette ligne de code a pour effet de créer un lien symbolique entre phpmyadmin et le serveur web.

Si cela ne suffit pas, vous pouvez tenter une désinstallation de Phpmyadmin, suivie d'une réinstallation, en veillant bien à préciser le mot de passe « root » que vous avez défini pour MariaDB lors de son installation.

<https://doc.ubuntu-fr.org/mysql>

Si le problème persiste, vous trouverez ci-dessous quelques conseils intéressants trouvés sur StackOverflow :

(source : <https://stackoverflow.com/questions/39281594/error-1698-28000-access-denied-for-user-rootlocalhost>)

The reason is that recent Ubuntu installation (maybe others also), mysql is using by default « the Unix auth socket plugin ».

Basically means that: dbUsers using it, will be "auth" by the system user credentials. You can see if your root user is set up like this by doing the following:

```
$ sudo mysql -u root # I had to use "sudo" since is new installation
```

```
mysql> USE mysql;
```

```
mysql> SELECT User, Host, plugin FROM mysql.user;
```

```
+-----+-----+
| User          | plugin          |
+-----+-----+
| root          | auth_socket     |
| mysql.sys     | mysql_native_password |
| debian-sys-maint | mysql_native_password |
+-----+-----+
```

As you can see in the query, the root user is using the auth_socket plugin

There are 2 ways to solve this:

1. You can set the root user to use the mysql_native_password plugin
2. You can create a new db_user with you system_user (recommended)

Option 1:

```
$ sudo mysql -u root # I had to use "sudo" since is new installation
mysql> USE mysql;
mysql> UPDATE user SET plugin='mysql_native_password' WHERE User='root';
mysql> FLUSH PRIVILEGES;
mysql> exit;
$ service mysql restart
```

Option 2: (replace YOUR_SYSTEM_USER with the username you have)

```
$ sudo mysql -u root # I had to use "sudo" since is new installation
mysql> USE mysql;
mysql> CREATE USER 'YOUR_SYSTEM_USER'@'localhost' IDENTIFIED BY '';
mysql> GRANT ALL PRIVILEGES ON * . * TO 'YOUR_SYSTEM_USER'@'localhost';
mysql> UPDATE user SET plugin='auth_socket' WHERE User='YOUR_SYSTEM_USER';
mysql> FLUSH PRIVILEGES;
mysql> exit;
$ service mysql restart
```

Remember that if you use option #2 you'll have to connect to mysql as your system username (mysql -u YOUR_SYSTEM_USER)

6 Changelog

Version 1.0 publiée le 08/12/2017 :

- première version forcément incomplète :(

Version 1.1 publiée le 15/12/2017 :

- correction de quelques anomalies, et notamment de liens internet perdus lors de la conversion de ce support de .doc vers .odt
- ajout de compléments d'infos dans la plupart des chapitres
- ajout d'un chapitre en annexe sur les problèmes de configuration du client SQL de Netbeans
- rédaction du chapitre 3.8 (techniques de pros)

Version 1.2 publiée le 22/12/2017 :

- ajout d'un chapitre relatif aux problèmes de configuration de Phpmyadmin
- rédaction des chapitres 3.9 et 4

Version 1.3 publiée le 2/01/2018 :

- rédaction du chapitre 3.9.1 consacré aux indexs
- complément à la fin du chapitre 3.9.4 sur les procédures stockées (sur la réutilisation de table temporaire après appel de la procédure) et correction d'une anomalie dans la dernière procédure (avec l'ajout d'un DROP TABLE IF EXISTS avant la création de la table temporaire).
- ajout de quelques références bibliographiques (dont article sur MariaDB publié dans GNU Linux Magazine)

Version 1.4 publiée le 6/01/2018 :

- ajout hors série des éditions Diamond consacré à MariaDB dans le chapitre bibliographie.

Version 1.5 publiée le 20/12/2018 :

- ajout du chapitre 3.8.4 sur la manipulation de données temporelles.