LES GUIDES DE
GNU LINUX MAGAZINE / FRANCE
HORS-SÉRIE N°88
France MÉTRO. : 12,90 € — CH : 18,00 CHF — BEL/PORT.CONT : 13,90 € — DOM TOM : 13,90 € — CAN : 18,00 $ CAD

CRÉEZ VOTRE BASE DE DONNÉES MYSQL MARIADB EN 5 ÉTAPES

ÉTAPE 1
Je conçois ma base de données

ÉTAPE 2
J'installe mon SGBDR MySQL MariaDB

ÉTAPE 3
J'utilise le langage SQL pour accéder à ma base

ÉTAPE 4
Je sécurise ma base et l'accès à mes données

ÉTAPE 5
Je crée des programmes utilisant mes bases de données

Édité par Les Éditions Diamond
L 15066 - 88 H - F: 12,90 € - RD
www.ed-diamond.com

Retrouvez toutes nos publications
LES ÉDITIONS DIAMOND
sur **www.ed-diamond.com**

GNU/Linux Magazine Hors-Série
est édité par **Les Éditions Diamond**

10, Place de la Cathédrale - 68000 Colmar - France

Tél. : 03 67 10 00 20 / **Fax :** 03 67 10 00 21

E-mail : cial@ed-diamond.com
lecteurs@gnulinuxmag.com

Service commercial : abo@gnulinuxmag.com

Sites : http://www.gnulinuxmag.com
http://www.ed-diamond.com

Directeur de publication : Arnaud Metzler

Chef des rédactions : Denis Bodor

Rédacteur en chef : Tristan Colombo

Responsable Service Infographie : Kathrin Scali

Responsable publicité : Tél. : 03 67 10 00 27

Service abonnement : Tél. : 03 67 10 00 20

Impression : pva, Druck und Medien-Dienstleistungen GmbH, Landau, Allemagne

Distribution France :
(uniquement pour les dépositaires de presse)

MLP Réassort :
Plate-forme de Saint-Barthélemy-d'Anjou.
Tél. : 02 41 27 53 12
Plate-forme de Saint-Quentin-Fallavier.
Tél. : 04 74 82 63 04

Service des ventes :
Distri-médias : Tél. : 05 34 52 34 01

IMPRIMÉ en Allemagne - PRINTED in Germany

Dépôt légal : A parution

N° ISSN : 0183-0864

Commission Paritaire : K78 976

Périodicité : Bimestrielle

Prix de vente : 12,90 Euros

PRÉFACE

Vous avez un projet nécessitant le stockage et l'accès à des données ? Alors vous avez acheté le bon magazine pour vous aider à créer pas à pas votre base de données **MySQL/MariaDB** !

Les données représentent l'élément fondamental de tout système d'information, application un tant soit peu complexe ou projet de recherche (essayez d'identifier des gènes sans séquence génétique...). Pour conserver ces données et pouvoir y rechercher des informations plus simplement, les bases de données (ou plus précisément des Systèmes de Gestion de Base de Données, abrégés en SGBD) ont vu le jour.

Il existe de nombreux SGBD concurrents, certains adoptant une même logique et d'autres se basant sur des mécanismes totalement différents. Parmi les plus connus, et du coup les plus utilisés, nous pouvons citer MySQL, **PostgreSQL** et **Oracle**. En 2009, suite à des événements que vous trouverez relatés dans les prochaines pages, un *fork* de MySQL a été créé : MariaDB. Rassurez-vous, les deux projets sont encore très proches et devraient rester compatibles pour une bonne partie de leurs fonctionnalités. Ainsi pratiquement tout ce qui est écrit dans ce hors-série (ou ailleurs) sur MySQL fonctionne avec MariaDB et réciproquement (attention à la nuance du « pratiquement » tout de même...).

Puisqu'autant de SGBD co-existent, il a fallu faire un choix et nous avons donc retenu MySQL/MariaDB. Pourquoi ? Parce qu'historiquement il s'agit du SGBD *open source* le plus employé ; bon nombre de serveurs web sont encore bâtis sur le traditionnel modèle LAMP : **Linux Apache** MySQL **PHP**. De multiples projets web phares utilisent ce modèle par défaut et vous avez sans aucun doute déjà utilisé MySQL ou MariaDB sans forcément le savoir. Mais il y a une différence entre le fait d'être simple utilisateur ou administrateur de la base. En effet, par où commencer pour créer une base de données ? Ce n'est pas toujours très simple, aussi, pour vous aider, nous avons voulu ce hors-série très pratique en vous accompagnant dans la création de votre propre base de données en 5 étapes :

1. Comprendre le fonctionnement d'un SGBD et concevoir le diagramme qui représente la structure des données au sein de la base ;
2. Installer MariaDB ;
3. Utiliser la base à l'aide de requêtes ;
4. Mieux comprendre le fonctionnement des SGBD et notamment la sécurisation ;
5. Accéder aux données de la base depuis un langage de programmation.

Suivez ces étapes et... lancez-vous !

Tristan Colombo

Sommaire

GNU/Linux Magazine Hors-Série N°88

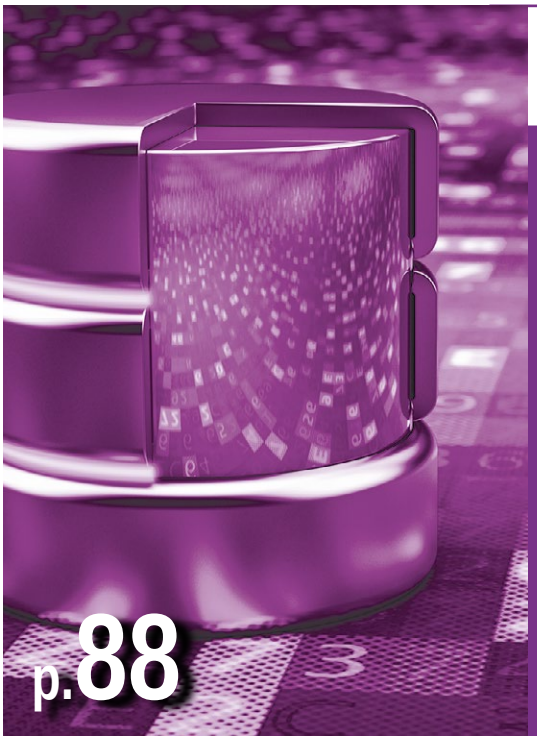
MYSQL MARIADB

ÉTAPE 3

J'UTILISE LE LANGAGE SQL POUR ACCÉDER À MA BASE

ÉTAPE 4

JE SÉCURISE MA BASE ET L'ACCÈS À MES DONNÉES



ÉTAPE 5

JE CRÉE DES PROGRAMMES UTILISANT MES BASES DE DONNÉES

ÉTAPE 1

1 JE CONÇOIS MA BASE DE DONNÉES

À découvrir dans cette partie...

1.1 Comprendre les bases de données relationnelles

Dans cet article, nous allons étudier les fondements du modèle relationnel et les règles qu'il faut respecter pour concevoir une bonne base de données. Nous verrons aussi que, parfois, il est pertinent de ne pas les respecter, nous expliquerons quand, pourquoi et comment. Les principes décrits dans cet article s'appliquent à l'ensemble des bases de données relationnelles et donc à des SGBDR (Système de Gestion de Bases de Données Relationnelles) tels que PostgreSQL, SQLite, Firebird, MS SQL Serveur, Oracle, DB2, etc. Les détails d'implémentation par contre peuvent différer. p. 08

1.2 Utiliser l'interface graphique MySQL Workbench

MySQL Workbench est une interface graphique pour Linux et Windows qui comprend un éditeur SQL complet pour développer, un modeleur de bases de données relationnelles, des outils de diagnostics de performance, des outils d'administration, etc. Nous allons présenter chacune de ces fonctionnalités qui en font aujourd'hui un outil indispensable pour travailler avec MySQL ou MariaDB. p. 22

ÉTAPE 1

JE CONÇOIS MA BASE DE DONNÉES

COMPRENDRE LES BASES DE DONNÉES RELATIONNELLES

Laurent NAVARRO

La base de données constitue les fondations pour de nombreuses applications, il est donc impératif qu'elle soit bien conçue. Les inventeurs du modèle relationnel ont établi des règles qu'il est généralement pertinent de respecter. Je vous invite à les découvrir.

Les bases de données relationnelles que nous utilisons sont fondées sur le modèle relationnel inventé par Edgar Frank Codd en 1970 (décrit dans la publication *« A Relational Model of Data for Large Shared Data Banks »*) et largement adopté depuis.

Ce modèle s'appuie sur l'organisation des données dans des relations (aussi appelées « entités », mais qui seront nommées « tables » dans la plupart des SGBDR), qui sont des tableaux à deux dimensions :

- les colonnes sont les attributs qui caractérisent la relation ;
- les lignes, aussi nommées « tuples », contiennent les données, et on les appelle « enregistrement » dans les SGBDR.

L'ordre des tuples n'a aucune importance ni signification.

1. CONCEPTION D'UNE BASE RELATIONNELLE

Je vous propose de nous mettre dans la situation où nous devons insérer dans une base de données le fichier Excel suivant géré par le responsable du personnel (qui est là, bien simplifié) :

Nom	Prénom	Service	Localisation
DUPOND	Marcel	Comptabilité	Toulouse
DURAND	Jacques	Production	Agen
LEGRAND	Denis	Ventes	Toulouse
MEUNIER	Paul	Production	Agen
MEUNIER	Paul	Achats	Agen
NEUVILLE	Henry	Ventes	Toulouse

Le modèle relationnel établit qu'il doit être possible d'identifier un tuple de façon unique à partir d'un attribut ou d'un ensemble d'attributs, lesquels constituent la clé de la relation.

Une clé est dite « naturelle » si un attribut ou un ensemble d'attributs la constituent. Dans l'exemple précédent, on constate qu'il est délicat d'en trouver une, car se pose le problème des homonymes qui, dans le pire des cas, pourraient travailler dans le même service.

Lorsqu'une clé naturelle n'existe pas, on introduit un nouvel attribut identifiant qui en fera office. Il s'agit, dans ce cas, d'une clé technique. Cet identifiant peut être un numéro affecté séquentiellement.

On recourt aussi à l'utilisation d'une clé technique quand la clé naturelle est trop grande ou pour des raisons techniques, par exemple des problèmes de changement de valeur de la clé lorsqu'il y a des relations maîtres/détails et que le SGBDR ne gère pas la mise à jour en cascade (le moteur **InnoDB** la supporte alors que le moteur **NDB** ne la gère pas).

On peut parfois avoir plusieurs clés pour identifier un tuple (c'est systématique si vous ajoutez une clé technique alors qu'il y a une clé naturelle). Toutes ces clés sont dites « candidates » et celle qui a été retenue pour identifier la relation est dite « primaire ».

L'absence de clé primaire dans une relation doit être exceptionnelle (moins de 1 % des tables). Lorsque rien ne s'y oppose, il faut privilégier l'utilisation de clé naturelle.

Nous introduisons donc une clé primaire technique avec la colonne **IdEmployé** de type numérique qui pourra être une colonne de type entier auto incrémenté dans MySQL.

IdEmployé	Nom	Prénom	Service	Localisation
5625	DUPOND	Marcel	Comptabilité	Toulouse
8541	DURAND	Jacques	Production	Agen
4521	LEGRAND	Denis	Ventes	Toulouse
8562	MEUNIER	Paul	Production	Agen
7852	MEUNIER	Paul	Achats	Agen
6214	NEUVILLE	Henry	Ventes	Toulouse

Un des grands intérêts du modèle relationnel est qu'il répartit l'ensemble des données dans différentes relations et établit des associations entre ces relations au moyen de leur clé primaire.

Ce principe permet d'éviter la duplication d'informations (dans notre exemple, les localisations associées aux services) et donc d'avoir des données plus consistantes. Nous allons donc obtenir les deux relations suivantes :

⇨ La relation **employé** :

IdEmployé	Nom	Prénom	IdService
5625	DUPOND	Marcel	100
8541	DURAND	Jacques	120
4521	LEGRAND	Denis	150
8562	MEUNIER	Paul	120
7852	MEUNIER	Paul	180
6214	NEUVILLE	Henry	150

⇨ La relation **service** :

IdService	Service	Localisation
100	Comptabilité	Toulouse
120	Production	Agen
150	Ventes	Toulouse
180	Achats	Agen

Nous allons profiter d'avoir réparti nos données dans 2 relations pour expliquer le concept de clé étrangère (*Foreign Key*). Dans notre exemple, nous allons dire que l'attribut **IdService** de la relation **employé** référence l'attribut **IdService** de la relation **service**. Les 2 attributs ont le même nom dans les 2 relations, mais ce n'est pas du tout obligatoire.

La mise en place d'une clé étrangère implique que dans l'attribut **IdService** de la relation **employé**, il ne sera possible de mettre que des valeurs présentes dans la relation **service**. Ceci implique donc qu'il ne sera pas possible de supprimer un service tant qu'il y a des employés qui y sont associés. Les clauses **ON DELETE** et **ON UPDATE** permettent cependant de modifier ce comportement, soit en coupant le lien avec l'option **SET NULL** qui mettra le champ à **NULL** dans la table **employé**, soit en propageant la mise à jour ou l'effacement avec **CASCADE**.

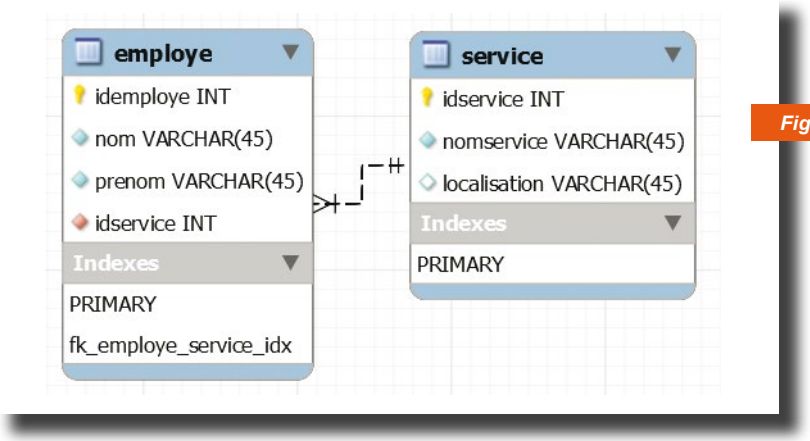


Figure 1

Diagramme de notre base de données dans MySQL Workbench.

Nous obtenons donc la base de données de la figure 1 si nous créons le modèle avec l'outil de modélisation de **MySQL Workbench**.

Cet outil nous générera un code SQL similaire à celui ci-dessous que j'ai un peu retouché.

Fichier

```
CREATE TABLE service (
  idservice INT NOT NULL AUTO_INCREMENT,
  nomservice VARCHAR(45) NOT NULL,
  localisation VARCHAR(45) NULL,
  PRIMARY KEY (idservice)
) ENGINE = InnoDB;

CREATE TABLE employe (
  idemploye INT NOT NULL AUTO_INCREMENT,
  nom VARCHAR(45) NOT NULL,
  prenom VARCHAR(45) NOT NULL,
  idservice INT NOT NULL,
  PRIMARY KEY (idemploye),
  CONSTRAINT fk_employe_service FOREIGN KEY (idservice) REFERENCES service
(idservice) ON DELETE NO ACTION ON UPDATE NO ACTION,
  INDEX fk_employe_service_idx (idservice ASC)
)ENGINE = InnoDB;
```

On y retrouve les deux commandes de création des tables avec nos colonnes. Les colonnes obligatoires sont spécifiées **`NOT NULL`** alors que les colonnes optionnelles sont spécifiées **`NULL`**, mais ce mot clé est optionnel.

J'ai supprimé les accents dans les noms des tables et des colonnes. MySQL les accepte ainsi que les espaces à condition de *backquoter* les noms, mais mon expérience m'a appris que ça pouvait être une source de problèmes insolubles significative dans certains outils clients. Je vous encourage donc à vous restreindre aux caractères alphanumériques et tirets bas pour tous vos noms, et de les spécifier sans les *backquoter*. Tout mettre en minuscule peut aussi, dans une moindre mesure, éviter quelques désagréments dans certains outils.

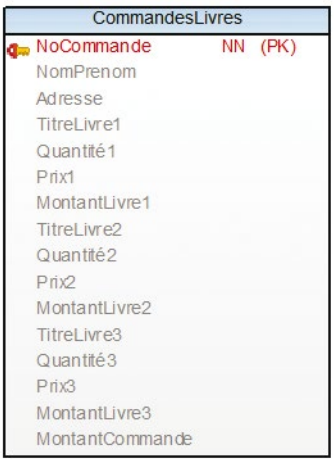
On remarque la déclaration de la FOREIGN KEY dans la table **`employe`**. Cette contrainte est nommée, ce qui est une bonne pratique et permet d'avoir des messages d'erreur plus explicites en cas de tentative de violation de la contrainte. La FOREIGN KEY ne peut être mise en place que lorsque la table référencée est créée, ceci implique que l'ordre de

création des tables a une importance. On remarque aussi qu'il a été mis un index sur la colonne `employe.idservice` : c'est une bonne pratique que nous discuterons plus tard dans l'article consacré aux index.

Nous venons donc de voir quelques notions fondamentales qui sont :

- ⇨ les clés primaires ;
- ⇨ les clés étrangères ;
- ⇨ les colonnes obligatoires.

Figure 2



Structure de notre fichier initial.

2. NORMALISATION

On entend souvent parler de base normalisée et dénormalisée. Je vous propose donc, maintenant que vous avez vu les principes clés sur un exemple simple, d'approfondir cette notion au travers d'un autre exemple un peu plus complexe qui est une base de gestion d'une librairie.

L'état initial est un fichier Excel (ou Calc) dont la structure simplifiée est présentée en figure 2. Eh oui, ça commence toujours par un fichier, voire parfois pire avec juste un cahier, mais ça devient assez rare.

2.1 Première forme normale (1NF)

Pour qu'une entité soit en première forme normale (1NF), elle doit :

- ⇨ avoir une clé primaire ;
- ⇨ être constituée de valeurs atomiques ;
- ⇨ ne pas contenir d'attributs ou d'ensembles d'attributs qui soient des collections de valeurs.

Un attribut qui a des valeurs atomiques n'est pas décomposable en sous-attributs. Par exemple, un attribut `NomPrenom` n'est pas considéré comme atomique, puisqu'il peut être décomposé en deux attributs : `Nom` et `Prenom`.

Le troisième point, « Ne pas contenir d'attributs ou d'ensembles d'attributs qui soient des collections de valeurs », signifie que :

- ⇨ il ne doit pas y avoir d'attributs qui contiennent, en fait, plusieurs valeurs séparées par des caractères comme des espaces ou des virgules tels que `"Dupond,Durand,Leclerc"` ;
- ⇨ il ne doit pas y avoir d'ensembles d'attributs qui soient en fait des listes de valeurs, tels que les attributs `TitreLivre1, TitreLivre2, TitreLivre3`, etc. de notre exemple.

De tels attributs doivent être mis dans une entité séparée qui aura une association de type `1-N` avec celle qui contenait ces attributs.

Mettons en pratique la première forme normale sur notre base exemple. Nous allons créer une entité des lignes de commandes pour résoudre le problème des collections sur les attributs (`TitreLivre`, `Quantité`, `Prix`) et atomiser les attributs `NomPrenom` et `Adresse`. Cela donnera le résultat de la figure 3 qui peut, raisonnablement, être considéré comme 1NF. Nous profitons de cette étape pour enrichir un peu notre modèle en ajoutant l'attribut ISBN et quelques autres champs.

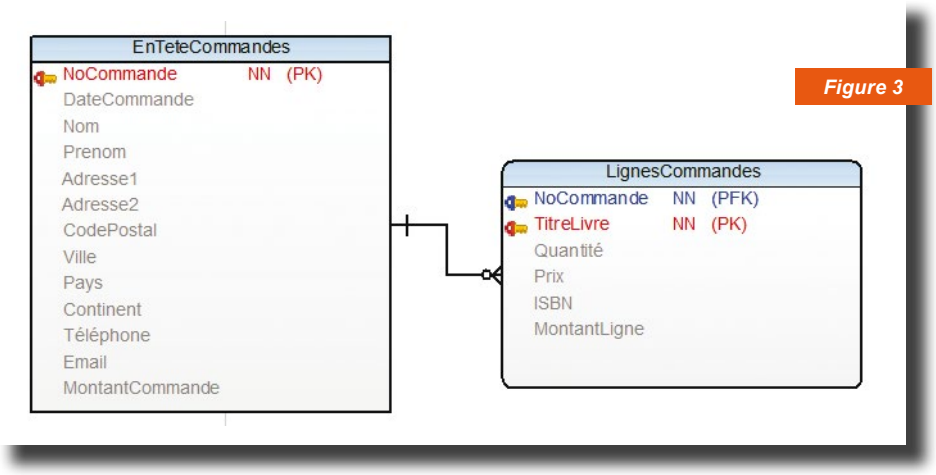


Figure 3

Structure de notre base en 1NF.

On parle ici de relation **1-N** entre l'entête et les lignes, car pour une ligne dans l'entité **EnTeteCommandes**, il y aura **N** lignes dans l'entité **LignesCommandes**, **N** pouvant être **0** (même si ça n'a pas forcement du sens), **1** ou n'importe quelle valeur positive.

2.2 Deuxième forme normale (2NF)

Pour qu'une entité soit en deuxième forme normale (2NF), il faut :

- qu'elle soit en première forme normale (1NF) ;
- que tous les attributs ne faisant pas partie de ses clés dépendent des clés candidates complètes et non pas seulement d'une partie d'entre elles.

Cette forme ne peut poser de difficultés qu'aux entités ayant des clés composites (composées de plusieurs attributs). Si toutes les clés candidates sont simples et que l'entité est 1NF, alors l'entité est 2NF. Attention, la règle s'applique à toutes les clés candidates et pas seulement à la clé primaire.

En figure 3, l'entité **lignescommandes** n'est pas 2NF, car la clé est **NoCommande** plus **TitreLivre**. Or, les attributs **Prix** et **ISBN** ne dépendent que de l'attribut **TitreLivre**, donc, seulement d'une partie de la clé. Pour transformer ce modèle en deuxième forme normale,

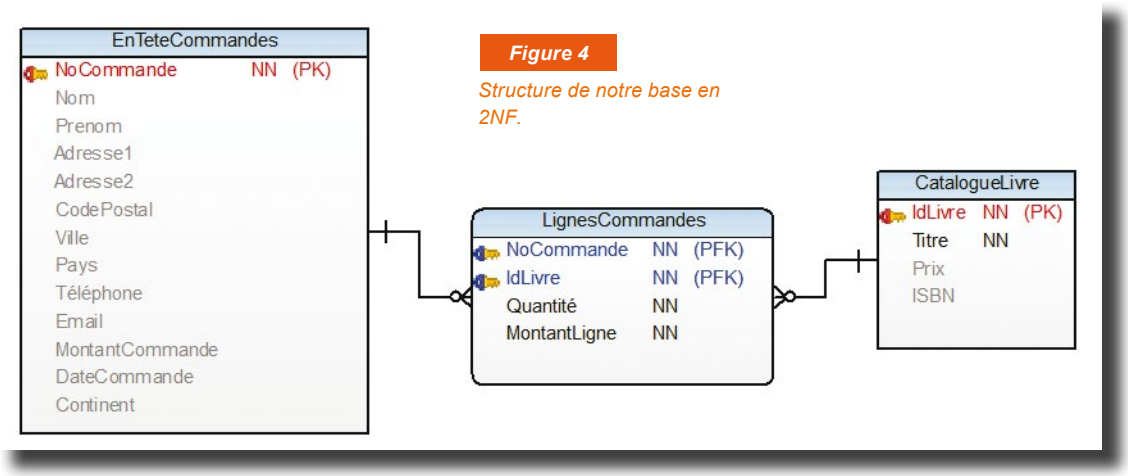


Figure 4

Structure de notre base en 2NF.

nous allons créer une nouvelle entité qui contiendra les informations relatives aux livres. L'application de cette forme normale permet de supprimer les inconsistances possibles entre les titres d'un même livre qui aurait été commandé plusieurs fois, car le modèle précédent permettait d'avoir des valeurs de **TitreLivre** différentes pour un même livre.

Nous introduisons un attribut **IDLivre** qui est quand même plus commun que d'utiliser le titre du livre comme clé et autorisera la modification de la valeur du titre au fil du temps si nécessaire.

Ceci donne le modèle de la figure 4 (page précédente) qui contient maintenant 3 tables.

2.3 Troisième forme normale (3NF)

Pour qu'une entité soit en troisième forme normale (3NF), il faut :

⇨ qu'elle soit en deuxième forme normale (2NF) ;

⇨ que tous les attributs ne faisant pas partie de ses clés dépendent directement des clés candidates.

En Figure 4, l'entité **EnTeteCommandes** n'est pas en 3NF, car toutes les informations relatives au client ne sont pas dépendantes de la commande (sauf si on considère que les clients ne reviennent jamais). Il faut donc introduire une nouvelle entité **clients**, qui contiendra les informations relatives aux clients.

Il devrait donc rester dans l'entité **EnTeteCommandes** les attributs **DateCommande** et **MontantCommande**.

Mais est-ce que l'attribut **MontantCommande** dépend directement de la clé ? Apparemment oui, si on considère seulement l'entité **EnTeteCommandes**.

Par contre, si on considère aussi ses associations, alors on s'aperçoit que cet attribut est en fait la somme des **MontantLigne** et ne dépend donc pas seulement de la clé, mais de l'entité **LignesCommandes**. C'est ce qu'on appelle un « attribut dérivé », c'est une redondance d'informations. Pour être en 3NF, il ne doit pas y avoir d'attributs dérivés, il faut donc supprimer l'attribut **MontantCommande**. Les attributs calculés à partir des colonnes de la même entité sont eux aussi des attributs dérivés et ne sont pas admis dans la troisième forme normale.

On pourrait faire la même remarque pour l'attribut **MontantLigne** de l'entité **LignesCommandes**. Cependant, il y a un argument qui l'autorise à persister : l'attribut **MontantLigne** est égal à **Quantité** x **Prix** du livre au moment de la commande,or, l'attribut **Prix** de l'entité **CatalogueLivre** est le prix actuel du livre. Ce prix est susceptible de changer dans le temps, l'information de prix intégrée dans l'attribut **MontantLigne** n'est donc pas la même information que le **Prix** de l'entité **CatalogueLivre**.

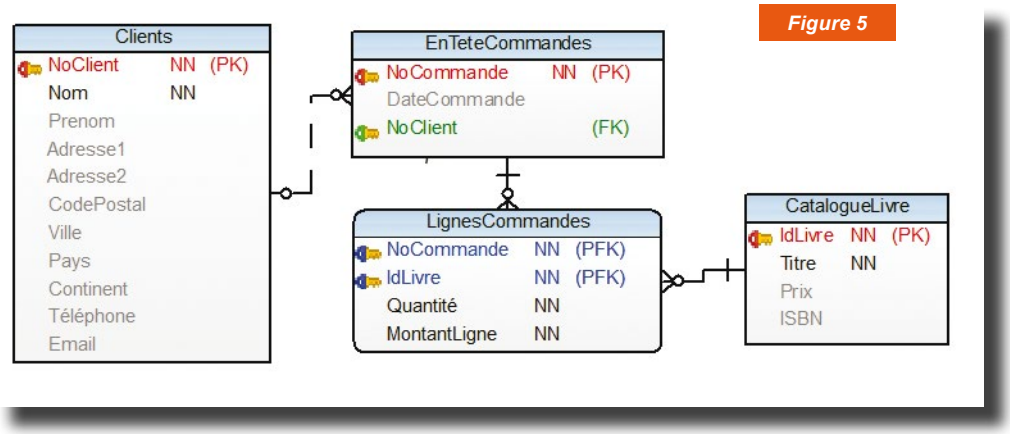


Figure 5

Structure de notre base en 3NF.

Ainsi, l'attribut **MontantLigne** n'est pas un attribut dérivé de l'attribut **Prix** de l'entité **CatalogueLivre** (voir la section sur la dénormalisation).

Ce qui nous donne le modèle de la figure 5 qui a maintenant 4 tables. On commence à avoir quelque chose de pas mal.

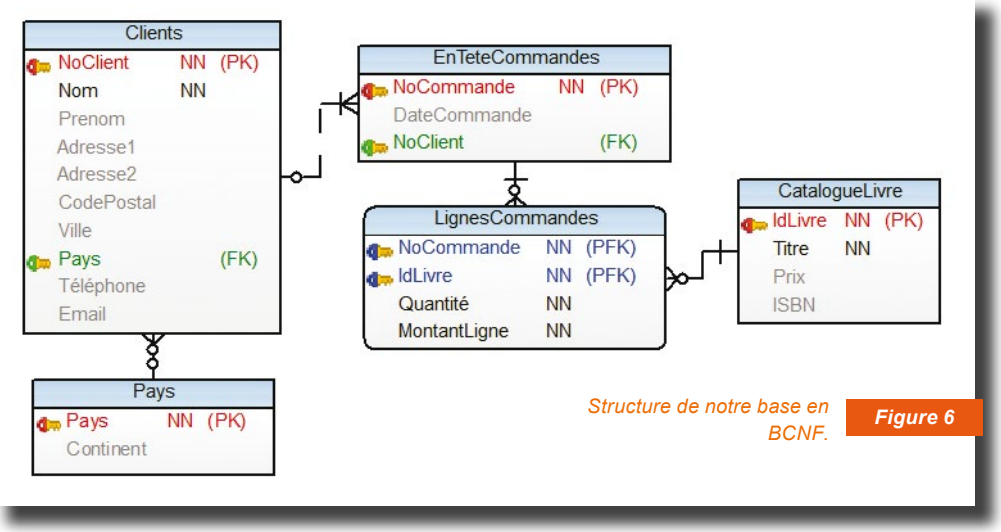
2.4 Forme normale de Boyce Codd (BCNF)

La forme normale de Boyce Codd est une version plus contraignante de la troisième forme normale.

Pour qu'une entité soit en forme normale de Boyce Codd (BCNF), il faut :

- qu'elle soit en troisième forme normale (3NF) ;
- que tous les attributs ne faisant pas partie de ses clés dépendent **exclusivement** des clés candidates.

En figure 5, l'entité **clients** n'est pas en BCNF, car l'attribut **Continent** dépend certes du client, mais il dépend surtout de l'attribut **Pays**. Pour être conformes, nous devons créer une nouvelle entité **Pays** permettant d'associer chaque pays à son continent (voir figure 6).



Structure de notre base en BCNF. **Figure 6**

Il existe d'autres formes normales (quatrième, cinquième et sixième formes normales). Cependant, elles sont un peu plus spécifiques. Tendre vers une troisième forme normale ou une forme normale de Boyce Codd est déjà un excellent objectif.

3. LA DÉNORMALISATION

La normalisation est une bonne pratique qu'il est plus que souhaitable d'appliquer.

Cependant, tout comme l'enfer est pavé de bonnes intentions, il faut savoir prendre un peu de recul par rapport aux règles afin de ne pas tomber dans certains excès. Si la normalisation fournit de bons guides, je pense qu'il faut savoir les remettre en cause s'il y a de bonnes raisons de le faire. Le plus important est de se poser les bonnes questions plutôt que d'appliquer aveuglément des règles. L'objet de cette section est de donner quelques règles pour ne pas respecter celles de la normalisation.

Attention !

Nous abordons ici la notion de dénormalisation. C'est-à-dire que nous allons étudier comment enfreindre certaines règles sur un modèle qui les appliquent.

Avant de dénormaliser un modèle, il faut d'abord le normaliser.

3.1 La dénormalisation pour historisation

Cette première forme de dénormalisation n'en est pas vraiment une. Nous l'avons déjà abordée, lors de l'étude de la troisième forme normale avec l'attribut **`MontantLigne`** de l'entité **`LignesCommandes`**.

Les formes normales ont pour objectif d'éviter la redondance de l'information. Si on conçoit le modèle avec une vision statique, on risque de passer à côté du fait que certaines valeurs vont évoluer dans le temps et il ne sera plus possible de retrouver après coup la valeur historique. Le prix courant d'un article et le prix de ce même article il y a six mois ne sont pas forcément les mêmes.

Comme nous l'avons déjà vu, il s'agit de données distinctes. Même si à certains moments les valeurs sont identiques, elles n'ont pas le même cycle de vie.

La méthode la plus commune pour gérer l'historique d'une valeur consiste à recopier l'information dans l'entité qui l'utilise. Une autre manière possible, et plus conforme à la logique des formes normales, pour gérer ce besoin est de conserver l'historique de la donnée référencée dans une entité séparée et horodatée plutôt que de répéter la donnée à chaque instance la référençant.

Si nous reprenons l'exemple du prix qui varie dans le temps, la première méthode possible consiste à recopier le prix dans chaque ligne de commande (voir diagramme de gauche de la figure 7). On peut aussi créer une entité contenant l'historique du prix du livre. Ainsi, la donnée prix n'est pas répétée et on pourra à tout moment la retrouver à partir de la date de la commande qui du coup devient obligatoire (voir diagramme de droite).

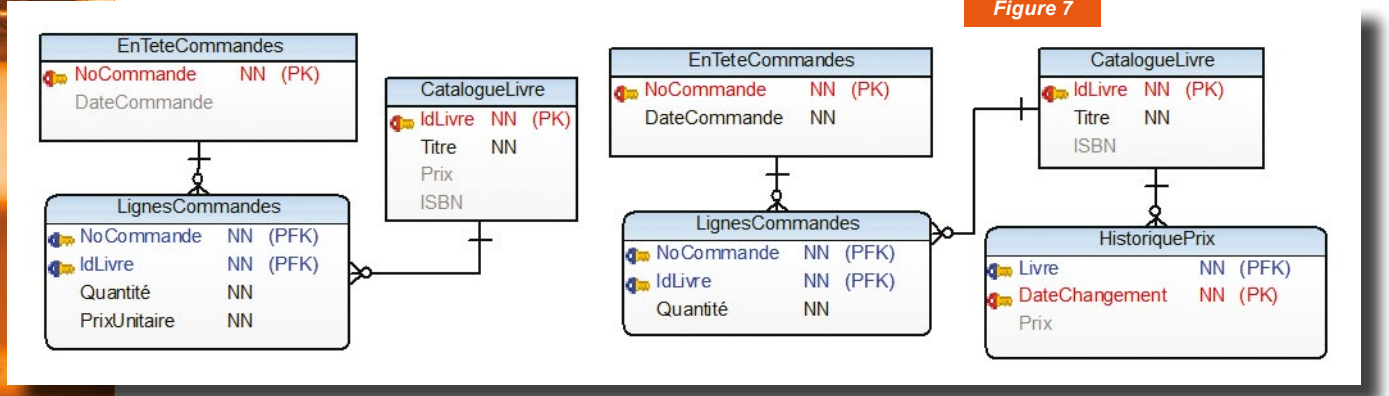
Dans le cas présent, mon cœur balancerait plutôt pour la première solution (à gauche), car je sais qu'en termes d'implémentation la seconde solution (à droite) sera plus compliquée. En effet, quand vous aurez besoin de connaître le prix d'une ligne de commande, vous devrez rechercher l'enregistrement d'historique le plus récent qui a une date antérieure à la date de la commande. Cette requête-là n'est déjà pas très simple. Si on l'ajoute le fait qu'on souhaite calculer le montant total des commandes d'un client pour une année, ça devient alors complexe et peu performant avec la seconde solution.

La première solution permet de répondre à cette requête de façon assez intuitive et efficace.

Dans d'autres contextes, la seconde solution pourrait présenter plus d'avantages que la première. Par exemple, si vous avez de nombreux attributs à garder en historique, qu'ils sont volumineux (grands textes descriptifs, photos, etc.) et évoluent peu au regard des commandes. La seconde solution permet aussi de connaître les variations de prix d'un produit même si celui-ci n'a pas été commandé entre les deux variations.

Comme toujours, il faut peser le pour et le contre de chaque solution plutôt que chercher forcément la solution mathématiquement la plus juste.

Figure 7



Deux options possibles pour historiser le prix d'un livre.

Certains disent qu'il ne faut pas dénormaliser pour arranger le développeur et qu'il faut toujours choisir la solution la plus propre sous peine de se traîner des « casseroles » pendant des années. C'est généralement vrai, cependant, comme le montre l'exemple précédent, l'application trop stricte de certaines règles peut conduire à inutilement complexifier une application et avoir des impacts substantiels sur les performances.

L'important est de se poser les bonnes questions et de faire vos choix en toute honnêteté intellectuelle.

3.2 La dénormalisation pour performance et simplification en environnement OLTP

La normalisation a pour but, entre autres, d'éviter la duplication de l'information afin de renforcer la consistance des données. Cela peut avoir pour effet de nuire aux performances. Comme nous l'avons vu, le modèle de droite de la figure 7 n'est pas adapté pour retrouver aisément l'ensemble des prix des lignes d'une commande.

Ainsi, lorsque vous avez besoin d'afficher sur une facture le montant total de la commande, il faut faire une requête compliquée.

Imaginons un instant qu'il y ait une politique, variable pour chaque article, de remises de prix en fonction de la quantité commandée. L'ajout d'un champ **MontantRemise** pourrait être considéré comme un non-respect des formes normales, car il dérive des attributs **Quantité** et **CodeArticle**. Cependant, je pense que, lorsqu'un attribut est une valeur qui est dérivée de façon complexe, on peut raisonnablement ne pas respecter à la lettre la troisième forme normale.

Abordons à présent le cas des attributs dérivés simples, mais qui rendent service tel que **MontantLigne** qui est égal à **Quantité** x **Prix** - **MontantRemise**. On ne peut pas dire que cela soit très compliqué à calculer, du coup ajouter cet attribut n'est qu'à moitié raisonnable. Une solution possible, et à mon avis fidèle à l'esprit des formes normales, est d'ajouter une colonne virtuelle (ou calculée suivant le SGBDR), laquelle contient le calcul et est maintenue par le SGBDR. Implémentée ainsi, il est sûr que cette information est consistante avec les informations dont elle dérive et est donc fidèle à l'esprit de la normalisation.

Exemple de concaténation des colonnes **nom** et **prenom** de notre premier exemple avec MySQL :

Fichier

```
alter table employe add nomprenom varchar(91)
    as ( concat(nom,' ',ifnull(prenom,'')) ) virtual;
```

La colonne **nomprenom** est à présent disponible, mais on est sûr qu'elle est bien à jour et en plus elle ne prend pas de place et peut même être indexée depuis la version 5.7.8 avec le moteur InnoDB.

La limite de ce type de champ est qu'il ne peut faire des calculs que sur les champs d'une même table.

Regardons maintenant un autre cas d'attributs dérivés simples : les attributs dérivés par agrégation, tels que l'attribut **MontantCommande** qui est la somme des **MontantLigne** du modèle de la figure 4. Sa présence ne respecte pas la troisième forme normale, mais elle peut être très pratique pour calculer des indicateurs de chiffres d'affaires (CA) en

temps réel. En plus d'être pratique, cela sera plus performant que de faire la somme des lignes de chaque commande. Cependant, il faudra maintenir cette valeur, donc faire régulièrement une requête qui recalcule la somme de **MontantLigne** pour mettre à jour l'attribut **MontantCommande** à l'aide d'un *trigger* par exemple. Cela peut être acceptable pour le cas de **MontantCommande**, mais il ne faut pas trop dériver, sinon il y a un risque de devoir calculer trop de données agrégées à chaque modification d'une donnée ce qui, in fine, sera contre-productif.

En effet, afin de garantir la consistance de ces données, chaque fois que vous modifiez (ou insérez ou effacez) un enregistrement de **LignesCommandes**, vous devez recalculer les valeurs qui en dérivent. Par exemple, si vous n'avez pas été raisonnable, vous pourriez devoir calculer toutes les valeurs suivantes : CA par jour, CA par mois, CA par jour par Livre, CA par mois par Livre, CA par client par an, CA par mois par pays, CA par mois par continent, etc. Vous risquez donc, si vous abusez de la dénormalisation, de plomber sérieusement vos performances lors des mises à jour des données de votre base, alors que votre objectif était d'améliorer les performances de votre système. L'utilisation de vues matérialisées sur certains SGBDR (Oracle, SQL Serveur, mais pas MySQL) permet de maintenir ce genre de données de façon consistante et relativement performante, car les calculs sur ces objets sont faits par variations et non pas par le recalcul complet des totaux (solution qui pourrait être extrêmement coûteuse).

Exemple de calcul du champ agrégé **MontantCommande** à partir des mises à jour sur la table **LignesCommandes**. Il faut traiter les différentes opérations possibles sur la table d'où la nécessité de 3 *triggers* :

Fichier

```
drop trigger IF EXISTS trg_lignescommandes_insert ;
drop trigger IF EXISTS trg_lignescommandes_update ;
drop trigger IF EXISTS trg_lignescommandes_delete ;
delimiter /
CREATE TRIGGER trg_lignescommandes_insert
AFTER INSERT ON lignescommandes FOR EACH ROW
BEGIN
    UPDATE entetecommandes e SET e.montantcommande = ifnull(e.montantcommande,0)
      + NEW.montantligne WHERE e.nocommande = NEW.nocommande;
END;
/
CREATE TRIGGER trg_lignescommandes_update
AFTER update ON lignescommandes FOR EACH ROW
BEGIN
    UPDATE entetecommandes e SET e.montantcommande = ifnull(e.montantcommande,0)
      - OLD.montantligne WHERE e.nocommande = OLD.nocommande;
    UPDATE entetecommandes e SET e.montantcommande = ifnull(e.montantcommande,0)
      + NEW.montantligne WHERE e.nocommande = NEW.nocommande;
END;
/
CREATE TRIGGER trg_lignescommandes_delete
AFTER delete ON lignescommandes FOR EACH ROW
BEGIN
    UPDATE entetecommandes e SET e.montantcommande = ifnull(e.montantcommande,0)
      - OLD.montantligne WHERE e.nocommande = OLD.nocommande;
END;
/
```

Nous avons travaillé ici par propagation des variations, ce qui nous évite de recalculer la somme, mais qui n'est pas applicable aussi facilement pour d'autres calculs tels qu'une moyenne.

Si on voulait maintenir une donnée telle que le CA/Client/Mois ça serait un peu plus compliqué, car ça ne serait pas forcement un **update**, mais peut-être un **insert** si c'est la première commande du mois pour ce client (alors que dans le cas de **MontantCommande** nous avons la certitude que la commande existe grâce à la présence de la clé étrangère).

4. AUTRES RÈGLES

Il existe d'autres règles, relatives au nommage des entités et des attributs, que l'on retrouve dans certains guides de bonnes pratiques. Je ne les trouve personnellement pas toutes à mon goût, mais je vais vous en citer quelques-unes (ainsi que diverses variantes) que vous déciderez peut-être de faire vôtres.

- Les noms des tables doivent tous être au singulier (ou au pluriel).
- Toutes les tables doivent commencer par **T_**, les vues par **V_**, les triggers par **TRG_**, etc..
- Tous les noms doivent être en minuscules ou majuscules ou CamelCase ou séparer les sous-mots par des tirets bas (ex : **noclient**, **NOCLIENT**, **NoClient**, **no_client**).
- À chaque table, associer un trigramme et l'utiliser pour tout ce qui y fait référence.

Par exemple, **Clients** aura le trigramme **cli**, **EnTeteCommande** aura le trigramme **cmd** et on préfixera tous les champs de la table **Clients** par **cli_** (**cli_noclient**, **cli_nom**, etc.). Le nom de la FOREIGN KEY sera **FK_CMD_CLI**.

- Nommer les attributs des FOREIGN KEY par le même nom ou préfixer celui du référençant par le nom de l'entité que l'on référence, par exemple **clients_noclient**.
- Toujours nommer les champs identifiants **ID_xxx** ou **cle_nomtable**.

Vous en croiserez sûrement d'autres, il y a quelques bonnes idées à prendre.

CONCLUSION

Nous avons étudié de façon un peu formelle les règles de conception d'une base de données, même si elles ne sont pas immuables et qu'il faut savoir faire preuve de pragmatisme, elles constituent un bon guide à la conception d'une base de données.

POUR ALLER PLUS LOIN

Vous trouverez sur Internet plusieurs cours en français de conception de base de données. Ils sont plus ou moins directifs et plus ou moins réalistes, mais il y a souvent des choses intéressantes à y retenir.

Certains se positionnent dans le cadre d'une méthodologie plus globale telle que **Merise**.

Il y a aussi quelques livres intéressants ; ils peuvent être un peu anciens, mais seront toujours d'actualité. La façon de concevoir une base de données relationnelle n'a pas trop changé depuis 20 ans, ce sont surtout les SGBDR qui ont changé. ▉

UTILISER L'INTERFACE GRAPHIQUE MYSQL WORKBENCH

Camille Huot

MySQL Workbench est le couteau suisse qui vous permet de tout faire avec vos bases de données MySQL ou MariaDB. Que vous soyez développeur ou même administrateur de quelques bases de données, cet outil vous permettra rapidement et facilement d'effectuer vos tâches courantes.

MySQL Workbench est LA GUI (*Graphical User Interface*, interface graphique pour l'utilisateur) fournie également par **Oracle** pour facilement administrer et développer avec sa base de données **MySQL**. Même si d'autres applications existent, MySQL Workbench est un incontournable et bénéficiera généralement des nouvelles fonctionnalités de MySQL en avance par rapport à la concurrence.

1. PREMIERS PAS

Avant de pouvoir dévoiler tout le potentiel de MySQL Workbench, connectons-nous à (au moins) une base MySQL. Pour cela, déroulez le menu ***Database*** puis cliquez sur ***Connect to Database...***

L'écran suivant permet de saisir toutes les informations nécessaires pour nous connecter à une base MySQL. Selon les privilèges accordés à votre utilisateur, vous aurez accès à plus ou moins de fonctionnalités. Pour montrer toutes les possibilités, nous serons connectés en root dans le cadre de cet article : Figure 1.

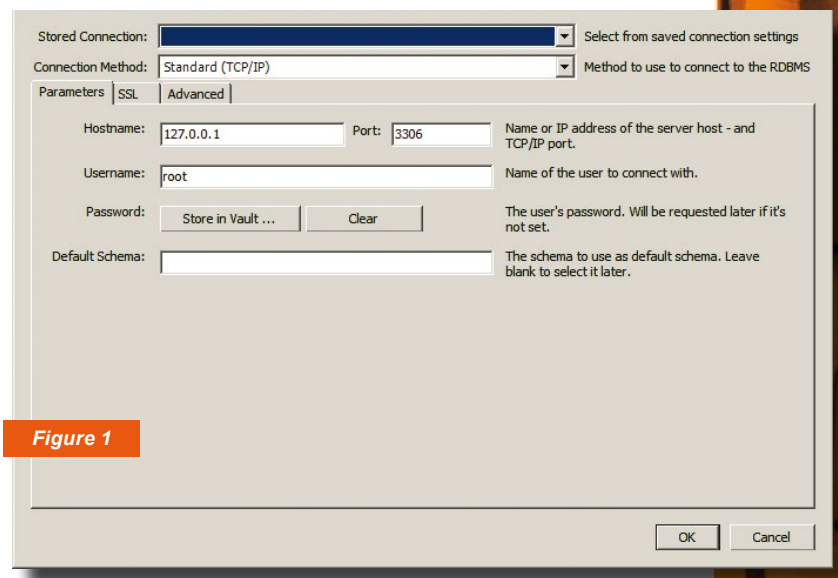


Figure 1

MySQL Workbench se connecte alors à la base de données et les différents menus s'affichent : Figure 2.

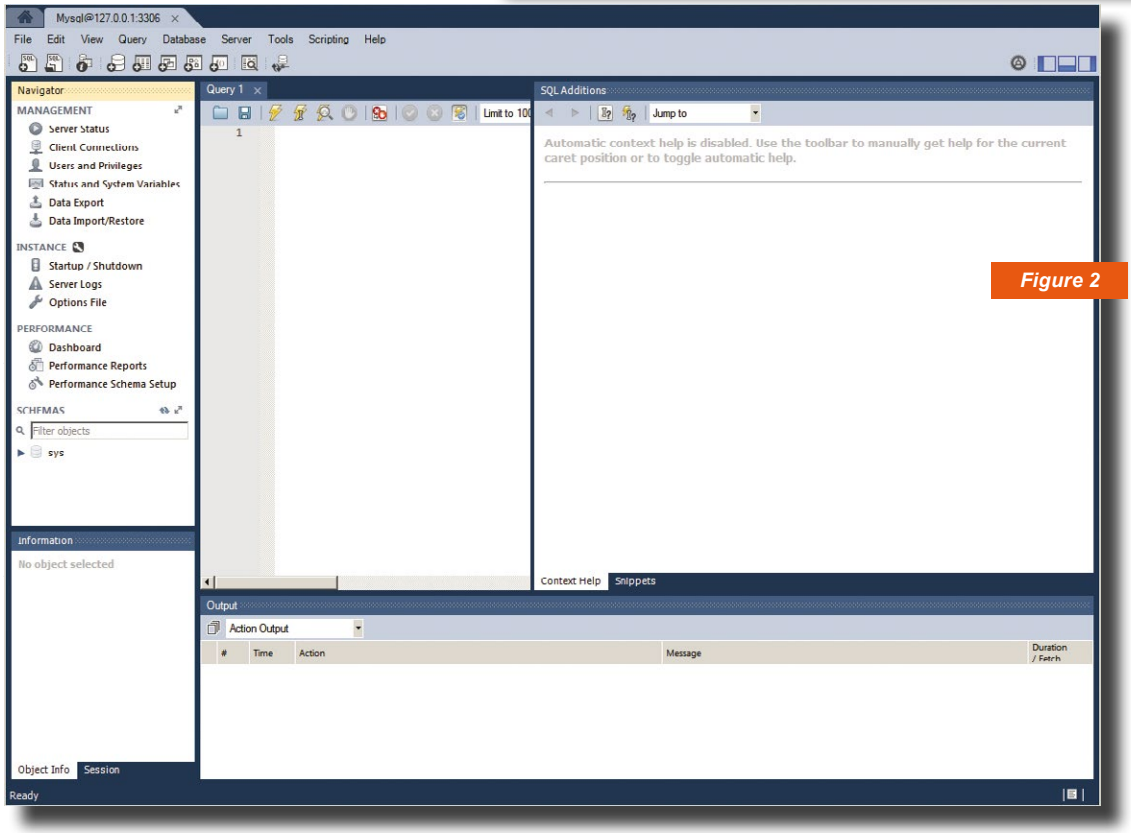


Figure 2

- Il permet d'avoir sous les yeux **plusieurs lignes de code**. Lors de votre phase de développement de requêtes ou pour des tâches simples d'administration, il fait un magnifique bloc-notes. Sélectionnez le code à exécuter pour n'envoyer qu'une partie au serveur.
- **Plusieurs fichiers** SQL peuvent être ouverts à la fois pour édition. Vous pouvez bien sûr les enregistrer et les ouvrir.
- La **coloration syntaxique** vous permettra de détecter immédiatement certaines erreurs de syntaxe ou de frappe.
- La **complétion** vous fera des propositions à partir des premiers caractères tapés.

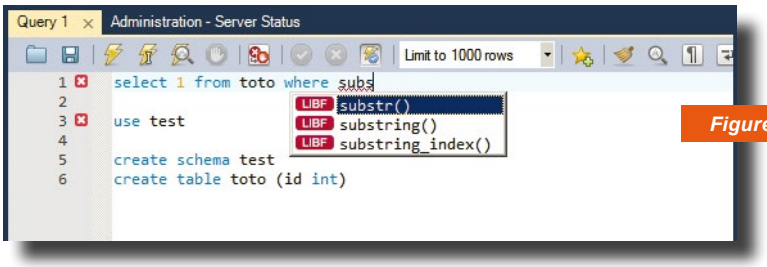


Figure 3

- Le résultat de la requête est facile à **sélectionner et copier** ailleurs, par exemple un tableur. Vous pouvez également **exporter un résultat au format CSV** par exemple.

3. GESTION DES DONNÉES

Le panneau sur la gauche ***Navigator*** présente l'accès aux différentes fonctionnalités étendues de MySQL Workbench. Le menu ***SCHEMAS*** permet de naviguer dans la structure des différents schémas auxquels vous avez accès. Vous pouvez ainsi facilement consulter la structure de vos tables, vos index, etc.

Consulter, mais aussi modifier, ajouter, supprimer des tables, colonnes, index, *triggers*...

La **génération du code SQL** nécessaire à la création d'une table existante s'avère un outil très puissant : Figure 4.

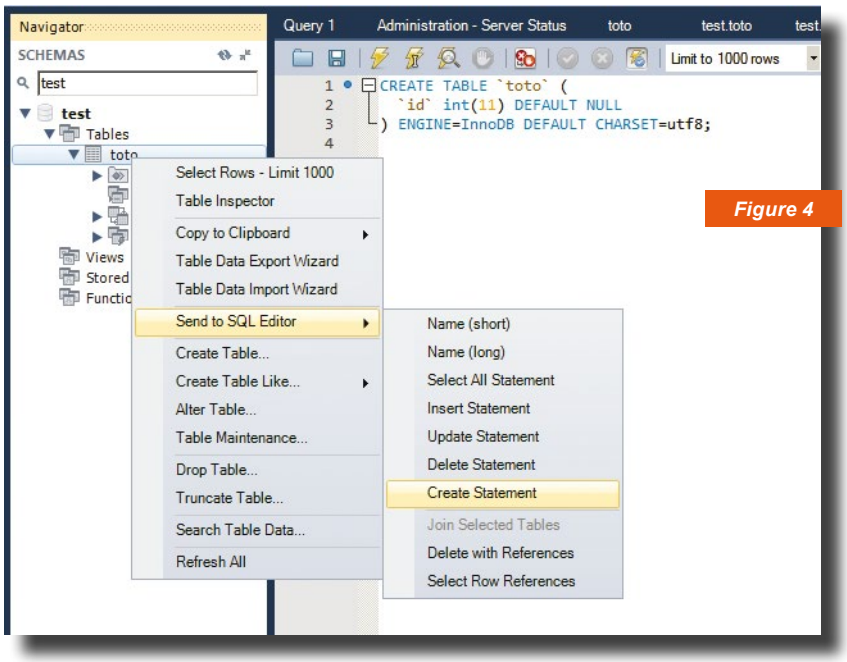


Figure 4

Vous pouvez ainsi vérifier avec exactitude les options qui sont utilisées par votre table (ici par exemple moteur de stockage **Innodb**, ensemble de caractères utf8, la colonne `id` a été créée en `DEFAULT NULL`, etc.) afin de recréer une autre table à l'identique (dans

un autre schéma ou une autre base). Mais vous pouvez aussi réutiliser ce code en le modifiant pour créer une autre table, les possibilités sont infinies.

Le menu contextuel qui se déroule aux différents niveaux de l'arborescence (schéma, table, colonne, vue, etc.) donne accès à tous les outils de visualisation (`select *`) ou de modification des données (`alter`). On peut également faire un export facilement.

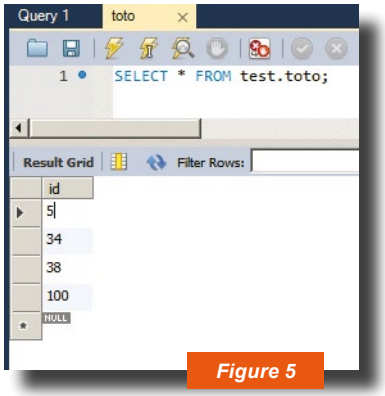


Figure 5

3.1 Éditer les tables directement

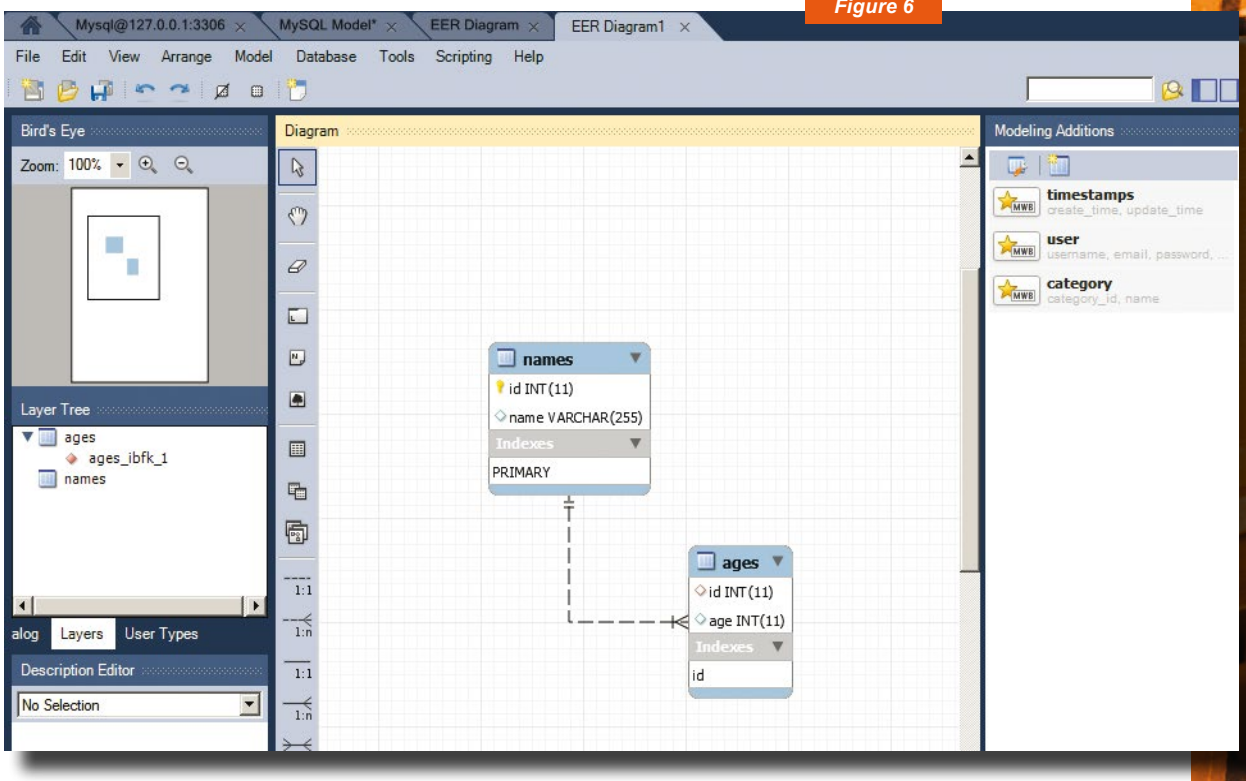
Lorsque vos tables ont une clef primaire, vous pouvez éditer directement le résultat d'un `SELECT` pour modifier les données de votre table : Figure 5.

4. MODÉLISER SES DONNÉES

Ne cherchez plus un outil de modélisation de données relationnelles, MySQL Workbench a tout ce qu'il faut pour rendre heureux !

On attaque directement avec le module de **rétroconception** (*Reverse Engineering*) qui permet de reproduire le modèle en partant d'un schéma existant. Pour l'utiliser, rendez-vous dans le menu ***Database*** et cliquez sur ***Reverse Engineer***. L'assistant s'ouvrira alors pour vous demander les informations de connexion et le nom du schéma à reproduire. Une fois terminé, le résultat est sous vos yeux : Figure 6.

Figure 6



Libre à vous ensuite de modifier le modèle pour le faire correspondre à vos besoins. Il est possible de regrouper les objets pour améliorer la lisibilité.

Une fois le modèle modifié, MySQL Workbench vous permet de **générer le script SQL** à exécuter pour mettre à jour vos tables de base de données (module de ***Forward Engineering***).

5. PERFORMANCE TUNING

Le module PERFORMANCE situé dans le *Navigator* vous offre quant à lui de nombreux outils de diagnostics et rapports de performance pour vous aider à améliorer les résultats de votre application.

5.1 Le Dashboard

Le *Dashboard* permet d'avoir une vue synthétique, rapide et visuelle de l'état de votre instance MySQL. Quelques graphiques (activité réseau, nombre de clients, nombre de requêtes, I/O InnoDB, etc.) permettent de repérer immédiatement un problème ponctuel ou une saturation.

À savoir

Ces informations de performances seront disponibles lorsque vous aurez activé le *performance_schema* (activé par défaut en 5.6.6+).

Figure 7

5.2 Performance Reports

Les rapports de performance pourront donner plus de détails au DBA pour comprendre (ou découvrir) le problème :

- Liste des fichiers de données ordonnés par nombre d'I/O ;
- Requêtes SQL les plus fréquentes, les plus longues à exécuter, celles qui renvoient beaucoup de lignes... ;
- Les jointures qui nécessitent la création de tables temporaires, mention spéciale pour les tables temporaires créées sur disque (ouch les performances !) ;
- Les requêtes qui font des lectures complètes (`FULL TABLE SCAN`) au lieu d'utiliser un index ;
- Les requêtes en erreurs.

Enfin, vous aurez également accès aux statistiques calculées pour vos tables ou index, ce qui permettra de comprendre le comportement de l'*optimizer*, voire de l'altérer (recalculer les stats, changer les index, mettre des *hints*...).

5.3 Plan et statistique d'exécution

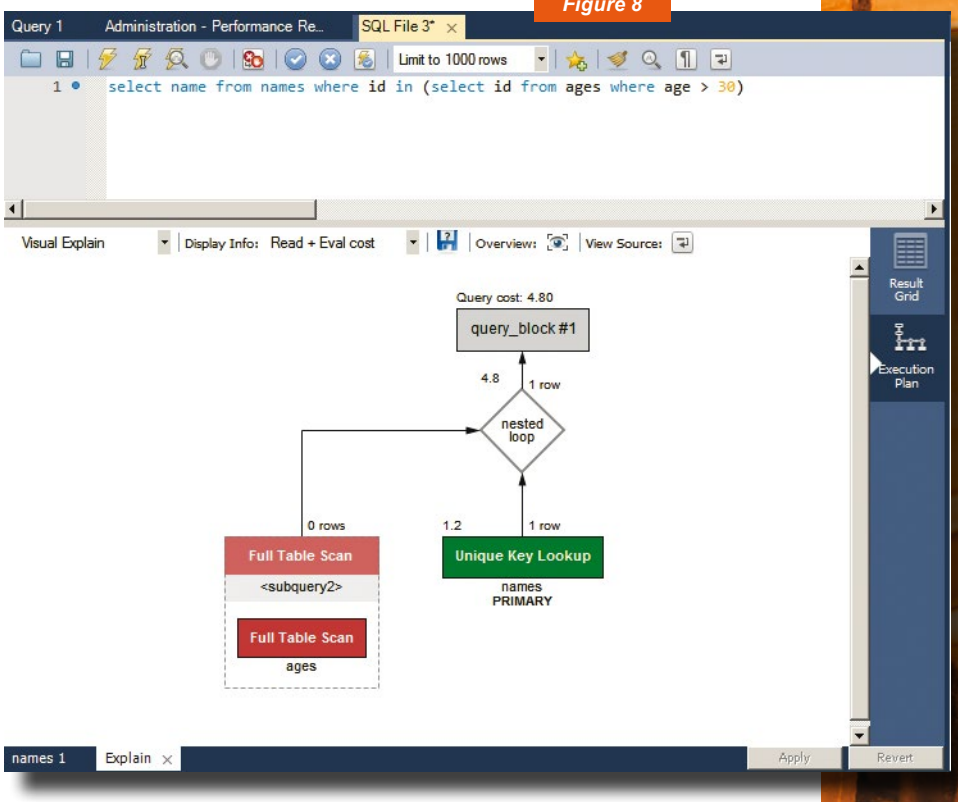
Connaître le plan d'exécution de sa requête est indispensable pour pouvoir l'optimiser. Il permet de savoir exactement ce que va faire MySQL pour compiler les résultats que vous avez demandés.

Les requêtes avec jointure, les sous-requêtes, les grosses tables, les index sont autant d'éléments qui peuvent faire « mal tourner » une requête.

L'éditeur SQL de MySQL Workbench permet de vous afficher le plan d'exécution de votre requête, ainsi que les statistiques qui ont permis d'en calculer le coût.

Tapez votre requête dans l'éditeur et au lieu de l'exécuter, faites ***Query*** > ***Explain Current Statement***.

Figure 8



Optimizer

L'*optimizer* est la partie de MySQL qui essaie de déterminer le moyen le plus rapide de lire les données pour vous retourner le résultat. Pour cela, il analyse les différentes possibilités : commencer par lire la table 1 et croiser avec la table 2, l'inverse, utiliser un index, ou pas... Tous les plans sont estimés par un coût et le moins cher est choisi.

6. ADMINISTRATION

Autant le dire tout de suite, MySQL Workbench n'est pas un outil d'administration et ses capacités sont assez limitées. Cependant, pour gérer quelques instances sans quitter sa souris, MySQL Workbench peut suffire.

Si vous utilisez Workbench pour accéder à un serveur MySQL distant, vous pourrez configurer un accès SSH ou Windows pour exécuter les commandes nécessaires à l'arrêt et au démarrage du serveur via le bouton ***Configure Server Management...*** de la fenêtre d'édition de la connexion Database.

MySQL doit être installé en tant que **service** pour pouvoir être administré (service Windows ou script `/etc/init.d/mysql`) et le compte utilisé doit avoir les droits pour gérer ce service.

Une fois paramétrée, l'interface d'administration de MySQL Workbench est simplement constituée d'un bouton ***Stop*** ou ***Start***, selon si l'instance est démarrée ou non.

Deux autres fonctionnalités sont assez pratiques :

- **La visualisation des logs de l'instance** : pratique pour voir rapidement ce qui se passe sans vous connecter vous-même et chercher puis ouvrir le fichier.
- La possibilité de **modifier toutes les variables de configuration** du `my.cnf`/`my.ini` via une interface à onglets reprenant tous les paramètres possibles. Modifiez les valeurs, sauvez la configuration puis redémarrez l'instance : Figure 9.

Note : Pour pouvoir effectuer ces tâches, MySQL Workbench aura besoin d'accéder directement au système de fichiers du serveur.

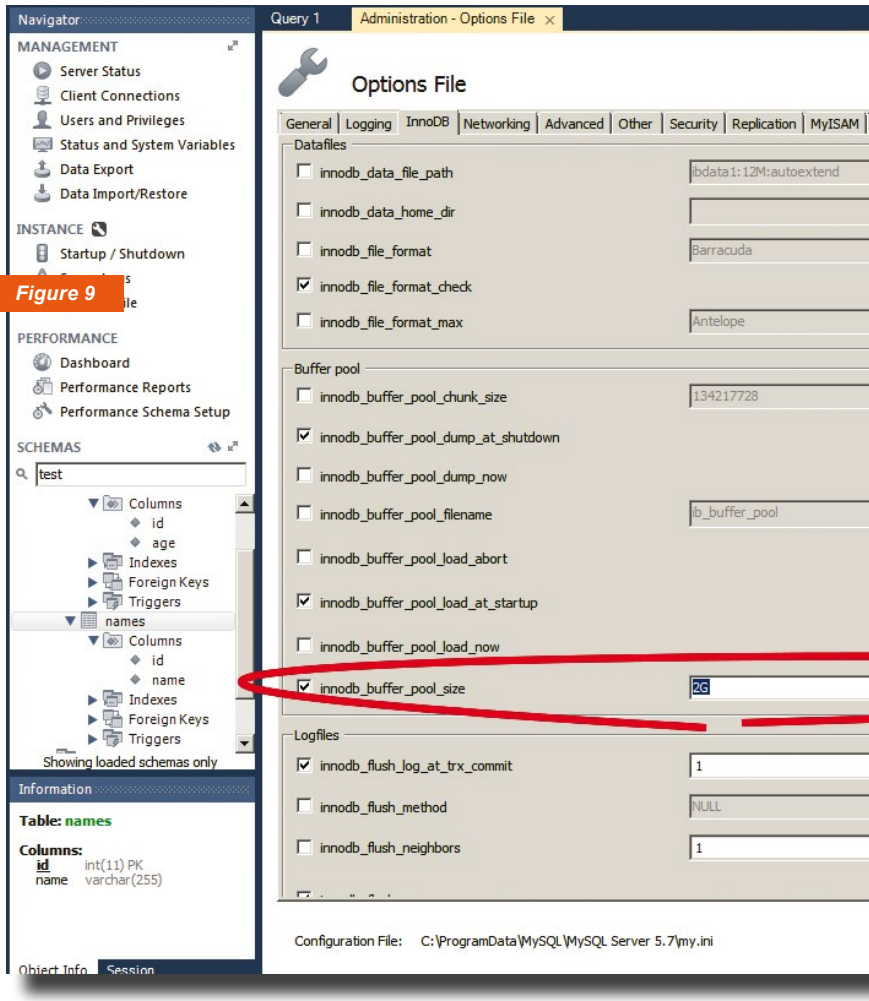


Figure 9

7. GESTION DES UTILISATEURS ET RÔLES

Une tâche d'administration souvent fastidieuse est la création d'utilisateurs et la mise à jour des privilèges.

Accédez à cet outil via ***Navigator*** > ***MANAGEMENT*** > ***Users and Privileges***.

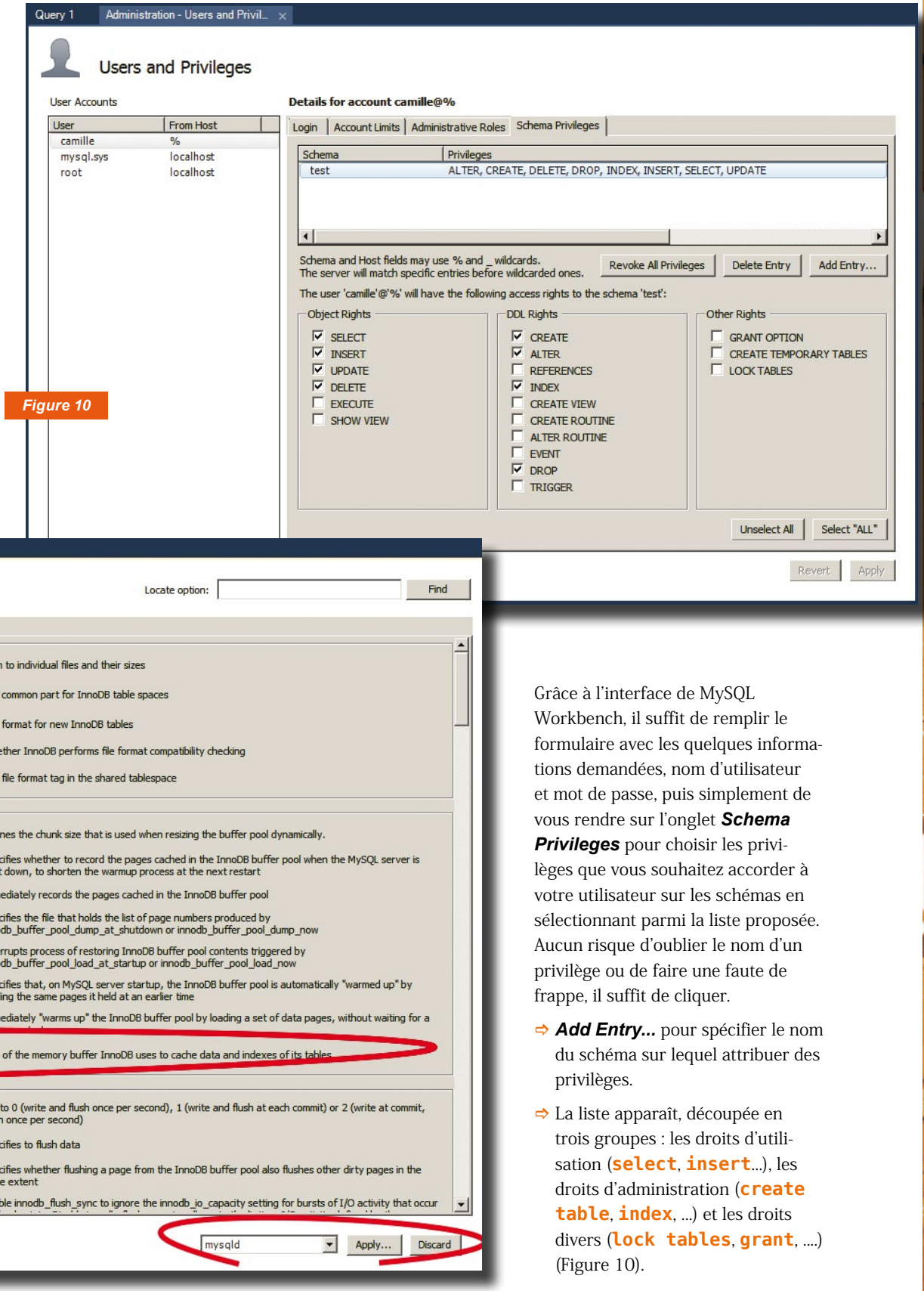


Figure 10

Grâce à l'interface de MySQL Workbench, il suffit de remplir le formulaire avec les quelques informations demandées, nom d'utilisateur et mot de passe, puis simplement de vous rendre sur l'onglet ***Schema Privileges*** pour choisir les privilèges que vous souhaitez accorder à votre utilisateur sur les schémas en sélectionnant parmi la liste proposée. Aucun risque d'oublier le nom d'un privilège ou de faire une faute de frappe, il suffit de cliquer.

- ***Add Entry...*** pour spécifier le nom du schéma sur lequel attribuer des privilèges.
- La liste apparaît, découpée en trois groupes : les droits d'utilisation (`select`, `insert`...), les droits d'administration (`create table`, `index`, ...) et les droits divers (`lock tables`, `grant`, ...) (Figure 10).

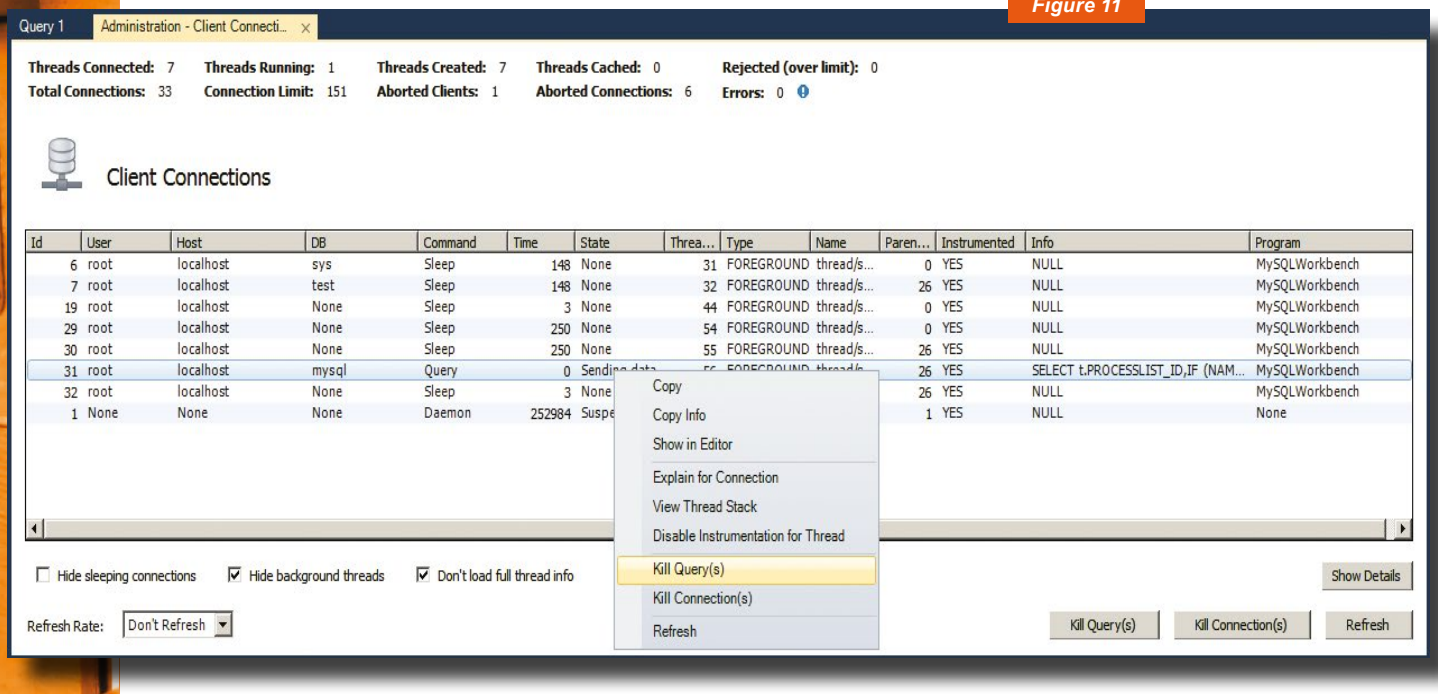
Si vous voulez donner des droits sur toute l'instance, par exemple pour configurer une réplication ou une sauvegarde, rendez-vous sur l'onglet ***Administrative Roles*** qui contient la liste des privilèges globaux.

8. REQUÊTES EN COURS, TUER UNE CONNEXION

Enfin, quoi de mieux pour savoir ce qu'il se passe sur votre instance MySQL que de lister les connexions ouvertes et les requêtes en cours ?

L'outil ***Client Connections*** (dans ***MANAGEMENT***) permet justement de savoir qui fait quoi et depuis combien de temps.

Figure 11



Si votre serveur de développement est soudainement lent, jetez un œil à cet écran pour repérer celui (ou ceux) qui ont une requête en cours depuis très longtemps (la durée figure en secondes dans la colonne ***Time***), examinez la requête en question (colonne ***Info*** et bouton ***Show Details*** pour avoir la requête complète, le plan d'exécution, etc.), trouvez des infos sur son auteur (colonnes ***User***, ***Host*** et ***Program***) et sévissez !

CONCLUSION

Nous avons passé en revue les fonctionnalités principales de MySQL Workbench. N'hésitez pas à prendre le temps d'essayer tous les menus et boutons pour voir ce qu'on peut faire, car cet outil est tout de même très fourni. ▮

C'est simple... c'est possible sur :

http://www.ed-diamond.com

ÉTAPE 2

2 J'INSTALLE MON SGBDR MYSQL MARIADB

À découvrir dans cette partie...

2.1 Choisir MySQL ou MariaDB et l'installer

MySQL et MariaDB sont les distributions les plus répandues du célèbre moteur de bases de données relationnelles libre. Nous expliquerons pourquoi il y a plusieurs distributions et comment installer MariaDB en utilisant les binaires précompilés fournis par MariaDB.com. Nous verrons également que plusieurs extensions sont disponibles pour faire des clusters MySQL. p. 34

ÉTAPE 2

J'INSTALLE MON SGBDR MYSQL MARIADB

CHOISIR MYSQL OU MARIADB ET L'INSTALLER

Camille Huot

MySQL ou MariaDB, que choisir ? Et surtout comment l'installer proprement et rapidement sans passer par le paquet de votre distribution qui vous « coincera » dans le choix de votre version.

1. QUELLES DIFFÉRENCES ENTRE MYSQL ET MARIADB ?

Réponse courte : c'est la même chose. D'ailleurs la commande pour lancer le client **MariaDB** est `mysql`.

Réponse longue : il était une fois en 1996...

1.1 La saga MySQL

MySQL est un système de gestion de bases de données relationnelles conçu originellement en Finlande par un certain Michael Widenius (entre autres). La première version publique a été publiée en 1996.

1.1.1 Un début fulgurant dans le monde du web PHP

Pendant l'essor d'Internet et du Web en particulier, avec la révolution **PHP**/MySQL, ce moteur de base de données gratuit connaît un franc succès et devient incontournable pour la majorité des nouveaux développeurs web. En effet, outre sa gratuité, le moteur de stockage **MYISAM** de MySQL fournit aux développeurs de très bonnes performances en sacrifiant seulement quelques fonctionnalités qui ne sont pas strictement nécessaires pour afficher un blog (pas de contraintes d'intégrité, pas de transaction, etc.) et en proposant notamment une réplication maître/esclave qui permet d'obtenir très facilement (et toujours gratuitement) un service en haute disponibilité de qualité professionnelle.

1.1.2 InnoDB propulse MySQL au rang des vrais systèmes de gestion de bases de données relationnelle

En 2001, le moteur de stockage **InnoDB**, alors développé par la société **Innobase**, est officiellement intégré dans la distribution MySQL. En effet, MySQL permet de déléguer la gestion physique des données à des moteurs de stockage, que chacun est libre de développer et de brancher à MySQL (comme un *plugin*). L'ajout d'InnoDB est une révolution, car il apporte tout ce qui manquait à MySQL pour devenir un concurrent direct des autres SGBDR (**Oracle Database**, **Microsoft SQL Server**, **IBM DB2**, etc.) à savoir notamment le support des transactions, des clefs étrangères, les verrous au niveau ligne, les contraintes d'intégrité, la journalisation (qui permet de récupérer automatiquement des données cohérentes après un crash) et la sauvegarde binaire à chaud.

Tout allait pour le mieux et de plus en plus de développeurs utilisaient MySQL pour stocker leurs données, même en dehors du Web. Puis les premiers doutes sur le futur du produit ont commencé à apparaître progressivement à partir de 2005, quand la société Oracle, qui développe le moteur de base de données éponyme fortement répandu en entreprise, a racheté Innobase, devenant ainsi propriétaire du code d'InnoDB. En effet, la communauté open source, très fière de la qualité et du succès de ce produit distribué sous licence GPL, craignait qu'Oracle ne le descende en plein vol.

1.1.3 De rachat en rachat

Finalement en 2008/2009, Michael Widenius a vendu MySQL à **Sun Microsystems**, ce qui pouvait sembler une bonne chose pour lui donner un peu de puissance commerciale. Sauf que Sun Microsystems a été racheté par Oracle quelques mois plus tard. La communauté est pétrifiée.

Ces longs mois d'incertitude concernant l'avenir de MySQL ont poussé une partie des développeurs à quitter Sun puis Oracle pour continuer le développement de MySQL (qui était sous licence GPL, donc un fork GPL est possible). Seuls le logo et le nom de MySQL étant protégés, Widenius a donc fondé la société MariaDB pour continuer son projet initial. **Percona** est une autre société qui s'est démarquée pendant cette période en reprenant le code d'InnoDB (open source lui aussi) sous le nom **XtraDB**.

1.1.4 MariaDB séduit

En 2009, la première version de MariaDB (livrée avec XtraDB au lieu d'InnoDB) est publiée. Oracle a fini par sortir de nouvelles versions de MySQL. Pour autant il est important de comprendre que la majorité du code source des deux produits est identique. Les quelques années qui ont suivi ont montré une préférence de la communauté Linux pour MariaDB. En effet, en plus de proposer les deux versions, certaines distributions comme **Fedora**, **Red Hat**, **Arch Linux**, **SuSE**, ou **Slackware** installent désormais MariaDB par défaut.

1.1.5 Une question de popularité ?

Enfin, aujourd'hui, comment choisir laquelle installer ? En production, vous allez certainement vous attacher à étudier les propositions commerciales concernant le type de support technique dont vous avez besoin et les options ou outils d'administration proposés. Sachant qu'a priori MySQL et MariaDB vont certainement continuer d'évoluer en parallèle très longtemps, sans être 100% équivalents, ni sans trop diverger. Sur des petits projets perso, il n'y a que très peu de chance que vous ayez besoin d'une fonctionnalité qui ne serait présente que sur l'une des versions.

Pour la petite histoire, My et Maria sont les prénoms des deux filles de Michael Widenius.

1.2 MySQL, MariaDB, MySQL Cluster, Galera… ?

Si MySQL et MariaDB sont grosso modo équivalents à quelques détails près, il existe des « extensions » comme **MySQL Cluster** ou **Galera Cluster** qui permettent de l'utiliser en mode cluster. Vous trouverez en fin d'article plus de détails sur ces versions spéciales.

Pour cet article, nous avons choisi MariaDB.

2. INSTALLER MYSQL/MARIADB

L'installation d'une base de données se fait en deux étapes distinctes.

1. On installe d'abord le logiciel ;
2. Puis on crée la base de données.

Cet article décrit l'installation manuelle des binaires de MariaDB 10.1 sur un système **Ubuntu** 12.04 LTS.

2.1 Pourquoi l'installation manuelle ?

⇨ Parce qu'on comprend mieux comment les choses fonctionnent quand on le fait soi-même (combien de personnes ont installé **Gentoo Linux** ne serait-ce que pour mieux comprendre son système GNU/Linux ?).

VOCABULAIRE

BASE DE DONNÉES : Le terme « base de données MariaDB » peut avoir un sens différent selon le contexte. Il est souvent employé d'une manière générique pour désigner l'ensemble des données servies par une instance de binaire MariaDB. Les termes suivants sont utilisés pour apporter plus de précision.

SCHÉMA ou DATABASE : Un schéma est un conteneur logique dans lequel les tables sont rangées afin de les isoler. Chaque nom de table doit être unique au sein d'un schéma. En général, on crée un schéma par application (ex. `wordpress`, `forum`, `paie`, etc.), ce qui permet de ne pas mélanger les différentes tables hébergées par un même MariaDB. Historiquement, MySQL utilisait le mot-clef `DATABASE` pour désigner un schéma, mais je vous déconseille son utilisation pour lever toute ambiguïté sur la définition d'une *database*. Utilisez le mot-clef synonyme `SCHEMA`.

DATADIR : C'est le répertoire principal dans lequel se trouvent tous les fichiers correspondant aux tables et index d'une base de données, eux-mêmes rangés dans des sous-répertoires correspondant aux schémas.

MOTEUR : Le moteur de base de données correspond à une installation contenant les binaires nécessaires au fonctionnement du serveur de base de données. Il s'agit en particulier du binaire `mysqld` et des bibliothèques associées. On peut avoir plusieurs moteurs, c'est-à-dire généralement plusieurs versions de MariaDB ou MySQL installées. Un seul moteur peut être utilisé pour plusieurs instances de bases de données.

INSTANCE ou SERVER : Une instance de base de données est un serveur actif (démarré) : un moteur `mysqld` est exécuté avec un fichier de configuration particulier (`my.cnf`), écoute sur un port réseau spécifique et permet l'accès aux schémas/tables d'un *datadir* spécifique. Par extension, on peut parler d'instance démarrée ou arrêtée, alors qu'en réalité l'instance cesse d'exister lorsque le processus `mysqld` est arrêté.

⇨ De plus, pour une utilisation en production (cela vaut également pour votre dédié perso) il est très important de maîtriser la version de son MySQL et de ne pas dépendre de la version choisie par les mainteneurs de votre distribution. Vous devez pouvoir changer de version si une nouvelle version corrige un bug qui vous affecte, proposer une nouvelle fonctionnalité que vous aimeriez utiliser ou simplement utiliser plusieurs versions simultanément.

Mais si vous désirez simplement tester MySQL, libre à vous d'utiliser votre gestionnaire de paquets (`apt-get`, `yum`, `emerge`, etc.).

Les plus aguerris pourront également compiler eux-mêmes MariaDB depuis le code source disponible également en libre téléchargement. En choisissant les paramètres de compilation soigneusement, vous obtiendrez un binaire optimisé pour votre processeur et vous pourrez activer ou désactiver individuellement les options.

2.2 Téléchargement de MariaDB

Butinez sur **https://downloads.mariadb.org/** et sélectionnez attentivement la version de MariaDB correspondant à votre système.

2.2.1 Sélectionner une version stable

Lorsqu'une branche de MariaDB est jugée suffisamment stable pour aller en production, celle-ci est taguée de « GA » pour *General Availability*. À moins que vous n'ayez besoin d'une fonctionnalité spécifique se trouvant uniquement dans une version plus récente (ou plus ancienne), choisissez toujours la dernière version GA de MariaDB. C'est également la version la mieux supportée par l'éditeur en cas de problème. Une fois qu'une branche est GA, de nouvelles versions seront toujours publiées, mais avec peu de changements pour ne pas perturber les applications déjà déployées. Surtout des correctifs.

Cliquez sur ***Download 10.1.x Stable Now!***.

2.2.2 Sélectionner l'architecture des binaires

Pour chaque version, de nombreuses distributions sont disponibles. Il y a le code source bien entendu, puis plusieurs distributions des binaires compilés, pour différents systèmes d'exploitation et architectures (**Windows**, **Solaris**, Linux x86, Linux x86_64, Linux PPC64...) et même pour plusieurs versions de GLIBC.

Utilisez les filtres de droite pour limiter le choix :

⇨ Operating System : ***Generic Linux*** ;

⇨ CPU : ***64-bit*** (si votre CPU/OS est 32-bit, choisissez 32-bit !).

La majorité de nos lecteurs voudront probablement une version **Linux x86_64** (ordinateurs à base de processeur Intel ou AMD) et les autres sauront certainement choisir la bonne par eux-mêmes ! Personnellement, j'utilise une machine virtuelle 32-bit pour me déplacer en formation MariaDB (oui, certaines sociétés ont encore leur bureautique sous Windows XP), donc je choisis Linux x86.

Selon votre distribution, la GLIBC installée sur votre système est différente. La GLIBC est une des bibliothèques principales d'un système Linux, qui définit les appels systèmes et des fonctions de base. Pour connaître sa version de GLIBC, le mieux est de demander à votre gestionnaire de paquets de vous l'indiquer. Le paquet se nomme généralement `libc6` ou `glibc`.

Sur Ubuntu :

Terminal

```
# dpkg -l libc6
||/ Name                 Version
Description
ii  libc6                2.15-0ubuntu10.6
Embedded GNU C Library: Shared librar
```

Exemple sur SUSE 11 :

Terminal

```
# rpm -q glibc
glibc-2.11.1-0.64.1
```

2.3 Décompresser l'archive

Vous aurez besoin d'environ 1,2 Gio d'espace libre pour décompresser l'archive que vous venez de télécharger. Nous installerons ces fichiers dans `/usr/local/`, qui sert justement à héberger les applications installées manuellement.

Astuce : si vous n'avez pas les droits root sur votre système, il est tout à fait possible d'installer MariaDB dans votre espace personnel. Lors de la configuration, il faudra simplement remplacer les différents chemins par défaut par des chemins sur lesquels vous avez les droits en écriture.

Terminal

```
# cd /usr/local/
/usr/local# tar xzf mariadb-10.1.18-linux-glibc_214-i686.tar.gz
/usr/local# du -sh mariadb-10.1.18-linux-glibc_214-i686
1.2G mariadb-10.1.18-linux-glibc_214-i686
```

Enfin, créez un lien symbolique pour masquer les détails de version du répertoire. Il vous facilitera la vie lorsque vous voudrez changer de version.

Note : si vous comptez lancer plusieurs instances de démon mysql, il est recommandé de créer un lien par instance :

Terminal

```
/usr/local# ln -s mariadb-10.1.18-linux-glibc_214-i686 mysql
```

Avant de pouvoir démarrer MariaDB, il nous faut encore créer un utilisateur et initialiser un *datadir*.

2.4 Créez un utilisateur Linux dédié pour mysql

Pour des raisons de sécurité, MariaDB ne tournera pas avec l'utilisateur root (on ne sait jamais). Par défaut, le démon `mysqld` lancé par root essaiera de passer sous l'utilisateur `mysql` pour fonctionner. Créons cet utilisateur :

Terminal

```
# useradd mysql
```

2.5 Créer et initialiser un datadir

Le *datadir* est le répertoire qui contiendra les données de votre instance MariaDB. Ce *datadir* contiendra :

- ⇨ les données de la base de données (tables, index, schémas, etc.) ;
- ⇨ les informations relatives à l'instance elle-même (configuration, *socket*, logs, etc.) ;
- ⇨ les fichiers directement créés et utilisés par les différentes fonctionnalités de MariaDB (réplication, *binary* logs, transaction logs InnoDB, cache Galera, etc.).

Attention : prévoyez suffisamment d'espace sur votre système de fichiers pour pouvoir contenir tous ces fichiers ! En production, la volumétrie dépassera rapidement le gigaoctet.

Par défaut, ce *datadir* est `/usr/local/mariadb-10.1.18-linux-glibc_214-i686/data/`. La plupart des distributions ont défini `/var/lib/mysql/`. Vous pouvez bien sûr choisir un autre endroit (paramètre `datadir` du `my.cnf` à adapter). Si vous avez besoin de plusieurs instances, chaque instance nécessitera son propre *datadir*. Plusieurs instances ne peuvent pas partager un *datadir*.

Nous allons utiliser `/var/lib/mysql/` :

Terminal
```
# mkdir /var/lib/mysql
# chown mysql:mysql /var/lib/mysql
```

Initialisation du *datadir* :

Terminal
```
# su - mysql
$ cd /usr/local/mysql
$ scripts/mysql_install_db --datadir=/var/lib/mysql
```

3. COMPLÉTER L'INSTALLATION ET PARAMÉTRER L'INSTANCE

Il est possible dès maintenant de démarrer une instance MariaDB qui permettra l'accès à notre nouvelle base de données sans fichier de configuration, avec l'utilisateur `root` ou directement `mysql` :

Terminal
```
# cd /usr/local/mysql
# support-files/mysql.server start --datadir=/var/lib/mysql
Starting MySQL
.161103 17:13:46 mysqld_safe Logging to '/var/lib/mysql/ubuntu-VirtualBox.err'.
 *
```

Mais afin de ne pas devoir spécifier le *datadir* à chaque lancement, nous allons procéder à quelques opérations post-installation.

3.1 Créer un my.cnf

Chaque instance MariaDB lit ses paramètres d'initialisation dans un fichier `my.cnf`. Avec une seule instance, le plus simple est de placer ce fichier dans `/etc/`.

NOTE

Pour configurer plusieurs instances, vous pouvez soit :

⇨ Utiliser la méthode « multi-mysql » :
Dans le même `my.cnf`, vous définirez les paramètres des différentes instances dans plusieurs sections `[mysqld1]`, `[mysqld2]`, etc.

⇨ Utiliser la méthode `MYSQL_HOME` :
Au démarrage, `mysqld` essaie de lire dans la variable d'environnement `MYSQL_HOME` (si elle est définie) le nom d'un répertoire dans lequel pourra se trouver un fichier `my.cnf` qui sera lu prioritairement par rapport à `/etc/my.cnf`. Ainsi, chacune de vos instances aura son `my.cnf` dans le répertoire de votre choix.

3.1.1 Modèles de my.cnf

MariaDB fournit des modèles (`my-small.cnf`, `my-medium.cnf`, `my-large.cnf`, `my-huge.cnf`, etc.) que vous pouvez copier directement dans `/etc` en fonction de la taille de votre base de données, de la mémoire disponible et du nombre de clients simultanés attendus :

Terminal
```
# cd /usr/local/mysql
# cp support-files/my-small.cnf /etc/my.cnf
```

Néanmoins une configuration simple ressemble à cela :

Fichier
```
[mysqld]
user=mysql
port=3306
datadir=/var/lib/mysql
```

Des centaines de paramètres existent pour optimiser la moindre parcelle de *buffer*. Les paramètres contenus dans les modèles sont généralement les plus utilisés (le plus important étant `innodb_buffer_pool_size` qui détermine combien de mémoire allouer à InnoDB : donnez-lui-en le maximum !).

3.2 Installer un script de démarrage automatique

Pour que MariaDB soit démarré automatiquement au démarrage, copiez simplement le script fourni dans `/etc/init.d/` et ajoutez-le aux services démarrés automatiquement :

Terminal
```
# cd /usr/local/mysql
# cp support-files/mysql.server /etc/init.d/mysqld
```

Pour Ubuntu :

Terminal
```
# update-rc.d mysqld defaults
```

Enfin, pour tester, arrêtez l'instance que nous avions démarrée précédemment et redémarrons-la sans argument :

Terminal
```
# /etc/init.d/mysqld stop --basedir=/var/lib/mysql
Shutting down MySQL..
*

# /etc/init.d/mysqld start
Starting MySQL
.161103 17:49:31 mysqld_safe Logging to '/var/lib/mysql/ubuntu-VirtualBox.err'.
*
```

Notre base MariaDB est fin prête à être utilisée en s'y connectant sur le port `3306` de votre serveur, si vous avez laissé le port par défaut.

MYSQL CLUSTER

MySQL Cluster permet depuis de nombreuses années de répartir vos données sur plusieurs serveurs (*sharding* automatique) tout en fournissant une disponibilité à 99,999% grâce à la réplication synchrone et à la tolérance aux pannes intégrée dans NBD, le moteur de stockage utilisé par MySQL Cluster.

Comment cela fonctionne ?

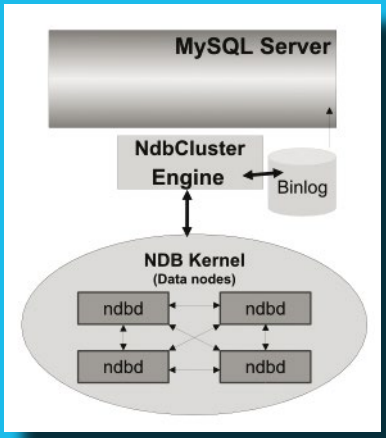
MySQL est modulaire. Lorsque vous créez une table ou ajoutez une ligne, ces opérations sont déléguées à un moteur de stockage qui est responsable d'enregistrer ces informations dans un fichier. Ainsi, les moteurs connus comme MYISAM et InnoDB reçoivent l'instruction d'ajout de ligne de la part du serveur MySQL et enregistrent les données de la ligne à leur façon.

Avec MySQL Cluster, c'est le moteur de stockage NDB qui s'occupe de cela. C'est en réalité NDB qui est une base de données répartie. Plusieurs serveurs NDB sont installés sur des serveurs différents, ce qui permet de fournir la tolérance de panne, et le moteur de stockage NDB de MySQL Cluster n'est qu'un client NDB qui va envoyer ses données au cluster NDB.

Exemple classique avec 4 nœuds

Avec 4 nœuds typiquement, les serveurs `ndbd` sont regroupés par 2 pour former un *replica set*. Les lignes de chacune de vos tables sont réparties sur les 4 serveurs NDB et chaque serveur `ndbd` va répliquer ses lignes sur l'autre `ndbd` de son groupe.

NDB étant un vrai moteur de base de données, il est possible de faire tourner plusieurs MySQL Cluster en même temps qui accèdent (en lecture/écriture) aux mêmes données, assurant ainsi la haute disponibilité promise.



4. SE CONNECTER, CRÉER DES SCHÉMAS ET DES UTILISATEURS, SÉCURISER

Le client `mysql` fourni avec la distribution MariaDB permet de se connecter à une instance et d'effectuer toutes les opérations possibles, en ligne de commandes.

L'installation par défaut n'est pas sécurisée. Nous pouvons nous y connecter sans mot de passe avec l'utilisateur anonyme (peu de privilèges) ou avec l'utilisateur `root` (administrateur).

Dans l'exemple suivant, nous nous connectons sans spécifier le nom de l'utilisateur. Le client essaiera d'utiliser `mysql`, car c'est le compte Linux que nous utilisons pour lancer le client. Cet utilisateur n'existant pas dans la base, nous serons connectés anonymement. Si vous lancez le client avec le compte `root`, vous serez connecté avec le compte root de MariaDB :

Terminal

```
# su - mysql
$ cd /usr/local/mysql
$ bin/mysql
Welcome to the MariaDB monitor.  Commands end with ; or \g.
Your MariaDB connection id is 5
Server version: 10.1.18-MariaDB MariaDB Server

Copyright (c) 2000, 2016, Oracle, MariaDB Corporation Ab and others.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

MariaDB [(none)]> show schemas;
+--------------------+
| Database           |
+--------------------+
| information_schema |
| test               |
+--------------------+
2 rows in set (0.00 sec)
MariaDB [(none)]> use test
Database changed
MariaDB [test]> create table toto (id int);
Query OK, 0 rows affected (0.03 sec)

MariaDB [test]>
```

Une fois connecté, nous pouvons lister les schémas (`show schemas`), nous placer dans test (`use test`) et créer une table (`create table`...).

4.1 Les utilisateurs et les privilèges

Comment fonctionnent les privilèges dans MariaDB ?

- Les tables qui contiennent vos données sont rangées dans des schémas.
- Les utilisateurs permettent d'identifier quels sont vos privilèges en fonction du *login* fourni et de l'adresse IP de votre client.
- Enfin, des privilèges sont attribués aux utilisateurs. Il existe différents niveaux de privilèges :
  - les privilèges sur une colonne d'une table ;
  - les privilèges sur une table ;
  - les privilèges sur un schéma (et l'ensemble des tables se trouvant dans le schéma) ;
  - les privilèges globaux (pour toute la base de données, notamment les privilèges d'administration).

4.2 Sécuriser une installation MariaDB

Comme nous l'avons vu précédemment, l'installation n'est pas sécurisée. Le script `mysql_secure_installation` permet de supprimer automatiquement les comptes anonymes et sans mot de passe :

Terminal

```
$ cd /var/lib/mysql
$ /usr/local/mysql/bin/mysql_secure_installation

NOTE: RUNNING ALL PARTS OF THIS SCRIPT IS RECOMMENDED FOR ALL MariaDB
      SERVERS IN PRODUCTION USE!  PLEASE READ EACH STEP CAREFULLY!
```

⇨ Étape 1 : Le script a besoin du mot de passe root actuel de MariaDB. Comme il n'y en a pas, faites juste <**Retrurn**> :

Terminal

```
In order to log into MariaDB to secure it, we'll need the current
password for the root user.  If you've just installed MariaDB, and
you haven't set the root password yet, the password will be blank,
so you should just press enter here.

Enter current password for root (enter for none):
OK, successfully used password, moving on...
```

⇨ Étape 2 : Il est temps de définir un mot de passe pour le compte root de votre base MariaDB. N'hésitez pas à être créatif et ne l'oubliez pas ! :)

Terminal

```
Setting the root password ensures that nobody can log into the MariaDB
root user without the proper authorisation.

Set root password? [Y/n]
New password:
Re-enter new password:
Password updated successfully!
Reloading privilege tables..
 ... Success!
```

⇨ Étape 3 : Autorisez le script à supprimer le compte anonyme (sauf si vous avez une bonne raison de ne pas le faire) :

Terminal

```
By default, a MariaDB installation has an anonymous user, allowing anyone
to log into MariaDB without having to have a user account created for
them.  This is intended only for testing, and to make the installation
go a bit smoother.  You should remove them before moving into a
production environment.

Remove anonymous users? [Y/n]
 ... Success!
```

⇨ Étape 4 : Restreindre l'accès via le compte root aux clients qui se connectent depuis le serveur uniquement. Cela empêche de prendre le contrôle à distance de votre base de données (ni vous ni un hacker). Avec un bon mot de passe, il est raisonnable d'accepter les connexions distantes, vous pouvez dire « n » :

Terminal

```
Normally, root should only be allowed to connect from 'localhost'.  This
ensures that someone cannot guess at the root password from the network.

Disallow root login remotely? [Y/n] n
 ... skipping.
```

⇨ Étape 5 : Supprimer le schéma **test** qui est créé par défaut. En production, il ne sera probablement pas utilisé, il est donc recommandé de le supprimer. Pour un serveur de test, vous pouvez le conserver :

Terminal

```
By default, MariaDB comes with a database named 'test' that anyone can
access.  This is also intended only for testing, and should be removed
before moving into a production environment.

Remove test database and access to it? [Y/n] n
 ... skipping.
```

⇨ Étape 6 : C'est terminé. Le script va recharger la table des privilèges pour qu'ils soient pris en compte et vous disposerez désormais d'une installation sécurisée :

Terminal

```
Reloading the privilege tables will ensure that all changes made so far
will take effect immediately.

Reload privilege tables now? [Y/n]
 ... Success!

Cleaning up...

All done!  If you've completed all of the above steps, your MariaDB
installation should now be secure.

Thanks for using MariaDB!
```

4.3 Créer des schémas et des utilisateurs

Nous pouvons maintenant en toute sécurité préparer les accès pour notre application.

Dans l'exemple suivant, une petite application de comptabilité a besoin d'une base MariaDB pour enregistrer ses données. Nous allons créer :

⇨ un schéma **compta** ;

⇨ un utilisateur **bigboss** avec tous les droits, c'est lui qui créera les tables, etc. ;

⇨ un utilisateur **comptable** avec les droits de modification des données pour les comptables ;

⇨ un utilisateur **lecture** avec des droits en lecture seule sur la table **OUTPUT** pour le module d'export de l'application :

Terminal

```
$ /usr/local/mysql/bin/mysql -uroot -p
CREATE SCHEMA compta;
CREATE USER bigboss IDENTIFIED BY 'CestMoiLeChef';
GRANT ALL PRIVILEGES ON compta.* TO bigboss;
CREATE USER comptable IDENTIFIED BY 'jaimeLesChiffres';
GRANT SELECT, UPDATE, INSERT, DELETE ON compta.* TO comptable;
CREATE USER lecture IDENTIFIED BY 'lecture000';
GRANT SELECT ON compta.OUTPUT TO lecture;
```

Vous aurez noté que la table `OUTPUT` n'existe pas encore. Essayez donc de vous connecter avec le compte `bigboss`, puis créez la table :

Terminal

```
CREATE TABLE OUTPUT (id int, out VARCHAR(255));
```

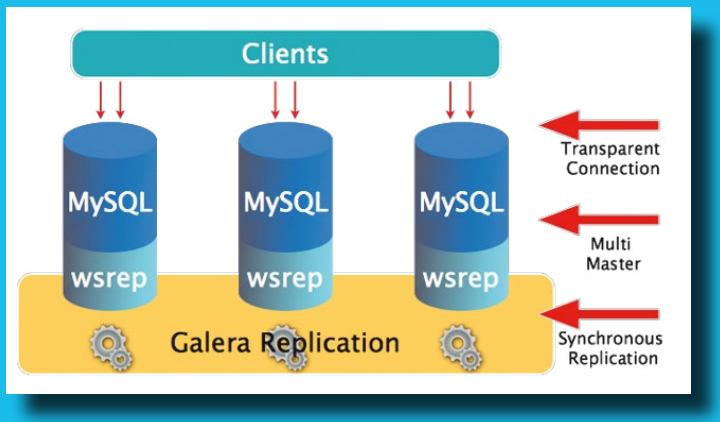
Votre base de données est fin prête pour accueillir le reste de ce hors-série ! Amusez-vous bien :) ▉

GALERA CLUSTER

Galera est une extension récente qui commence à être de plus en plus répandue. Elle est désormais incluse dans MariaDB. Oracle a annoncé une technologie similaire dans son MySQL 8.0.

Comment cela fonctionne ?

Il s'agit d'une extension qui permet à MariaDB, au moment où un client valide une modification (`commit`), de répliquer de manière synchrone cette modification à un groupe d'autres serveurs MariaDB, appelé Cluster Galera. Le mécanisme de synchronisation permet de vérifier que les verrous nécessaires à la transaction sont bien disponibles sur tous les nœuds, auquel cas la transaction est validée par tous les nœuds du *cluster*. Si un verrou utilisé par la transaction était occupé par une autre instance, alors la vérification échoue et la transaction n'est validée nulle part.



Ce mécanisme permet de fournir une réplication synchrone (les transactions validées sont présentes sur tous les nœuds du cluster) et donc une très bonne tolérance aux pannes (pas de delta à rattraper si un *master crashe*). De plus, il est possible d'écrire sur n'importe quel nœud du cluster (multi-master), ce qui simplifie énormément la gestion de la topologie (pas de reconfiguration à effectuer en cas d'incident) et permet d'améliorer la répartition de charge.

ÉTAPE 3

3 J'UTILISE LE LANGAGE SQL POUR ACCÉDER À MA BASE

À découvrir dans cette partie...

3.1 Utilisez correctement l'instruction Select pour vos requêtes

Le SQL est un langage qui a certes ses limites, mais qui a quand même plus de capacités que ses détracteurs voudraient le faire croire. Vous verrez dans cet article que sans atteindre la richesse d'un langage de programmation classique, il permet de faire pas mal de choses quand on parle de manipulation de données et qu'il mérite un peu d'investissement pour l'apprendre. p. 50

3.2 Augmentez les performances de vos bases de données avec les index

Les index font l'objet de mythes et de légendes, certaines personnes pensent que c'est compliqué à utiliser, d'autres pensent qu'il faut indexer toutes les colonnes. Les index sont un mécanisme assez efficace qui est assez simple à mettre œuvre. Il est important de comprendre comment ça marche afin de les utiliser à bon escient et d'en tirer pleinement profit. Cet article s'applique aux principaux SGBDR du marché (MySQL/MariaDB, PostgreSQL, Oracle, SQL Server...). p. 68

ÉTAPE 3

J'UTILISE LE LANGAGE SQL POUR ACCÉDER À MA BASE

UTILISEZ CORRECTEMENT L'INSTRUCTION SELECT POUR VOS REQUÊTES

Laurent NAVARRO

L'instruction select est l'instruction de base du SQL. Elle est malheureusement utilisée de façon trop basique par trop de gens. Je vous propose de l'étudier de façon un peu plus approfondie.

L'instruction **select** dans sa forme la plus commune est :

Fichier

```
SELECT colonne1,colonne2 as alias_colonne2, colonne3+colonne4 as colonnecalc1, ...
FROM table1, ...
WHERE des conditions
ORDER by des colonnes
```

Je vous propose aujourd'hui de vous montrer quelques façons d'écrire des requêtes un peu évoluées, toutes issues de besoins réels que j'ai croisés un jour ou l'autre. L'idée ici est de présenter des façons variées d'écrire des requêtes. L'objectif n'est pas de présenter la façon optimale d'écrire une requête, mais de montrer la variété dans les façons d'écrire du SQL, car en SQL comme dans les autres langages il y a plusieurs façons d'obtenir le même résultat.

Attention, cet article est assez dense, c'est un condensé de ce que vous trouveriez dans 100 pages d'un bouquin sur le SQL, mais avec beaucoup moins d'explications. Si vous découvrez ces notions, n'hésitez pas à prendre un peu de temps pour essayer les requêtes ou chercher des tutoriels sur le sujet.

Nous allons utiliser le modèle de données présenté à la figure 1, créé initialement par **Oracle** dans les années 80, bien avant qu'il ne rachète **MySQL**. Vous trouverez les scripts pour la recréer avec divers SGBDR sur Internet **[1]**.

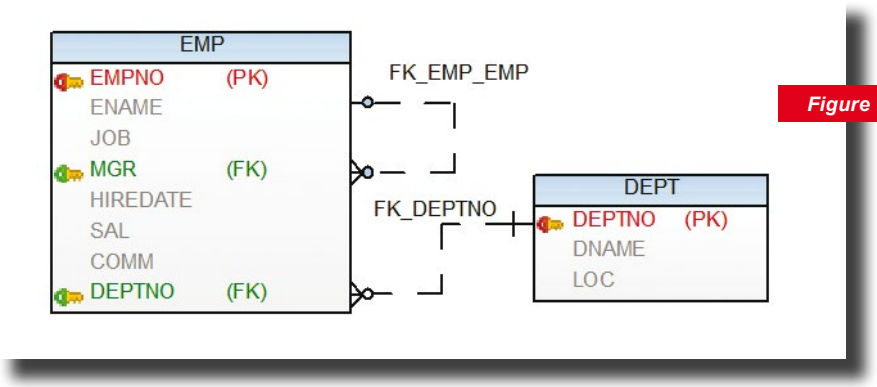


Figure 1

Modèle de données utilisé pour nos requêtes.

Il contient l'ensemble de données suivant pour la table **EMP** :

Terminal

```
+-------+--------+-----------+------+------------+------+------+--------+
| EMPNO | ENAME  | JOB       | MGR  | HIREDATE   | SAL  | COMM | DEPTNO |
+-------+--------+-----------+------+------------+------+------+--------+
|  7369 | SMITH  | CLERK     | 7902 | 1980-12-17 |  800 | NULL |     20 |
|  7499 | ALLEN  | SALESMAN  | 7698 | 1981-02-20 | 1600 |  300 |     30 |
|  7521 | WARD   | SALESMAN  | 7698 | 1981-02-22 | 1250 |  500 |     30 |
|  7566 | JONES  | MANAGER   | 7839 | 1981-04-02 | 2975 | NULL |     20 |
|  7654 | MARTIN | SALESMAN  | 7698 | 1981-09-28 | 1250 | 1400 |     30 |
|  7698 | BLAKE  | MANAGER   | 7839 | 1981-05-01 | 2850 | NULL |     30 |
|  7782 | CLARK  | MANAGER   | 7839 | 1981-06-09 | 2450 | NULL |     10 |
|  7788 | SCOTT  | ANALYST   | 7566 | 1987-04-19 | 3000 | NULL |     20 |
|  7839 | KING   | PRESIDENT | NULL | 1981-11-17 | 5000 | NULL |     10 |
|  7844 | TURNER | SALESMAN  | 7698 | 1981-09-08 | 1500 |    0 |     30 |
|  7876 | ADAMS  | CLERK     | 7788 | 1987-05-23 | 1100 | NULL |     20 |
|  7900 | JAMES  | CLERK     | 7698 | 1981-12-03 |  950 | NULL |     30 |
|  7902 | FORD   | ANALYST   | 7566 | 1981-12-03 | 3000 | NULL |     20 |
|  7934 | MILLER | CLERK     | 7782 | 1982-01-23 | 1300 | NULL |     10 |
+-------+--------+-----------+------+------------+------+------+--------+
```

Et pour la table **DEPT** :

Terminal

```
+--------+------------+----------+
| DEPTNO | DNAME      | LOC      |
+--------+------------+----------+
|     10 | ACCOUNTING | NEW YORK |
|     20 | RESEARCH   | DALLAS   |
|     30 | SALES      | CHICAGO  |
|     40 | OPERATIONS | BOSTON   |
+--------+------------+----------+
```

1. COLONNES CALCULÉES

Il est possible de créer des colonnes calculées directement dans vos requêtes.

Par exemple pour calculer le salaire augmenté de 10 % (**`*1.1`**) arrondi à l'unité (**`round(X,0)`**) nous écrirons cette requête :

Fichier

```
SELECT ename,job,sal,round(sal*1.1,0) SalPlus10Pct
FROM emp;
```

Ceci nous donnera le résultat suivant :

Terminal

```
+--------+-----------+------+--------------+
| ename  | job       | sal  | SalPlus10Pct |
+--------+-----------+------+--------------+
| ADAMS  | CLERK     | 1100 |         1210 |
| ALLEN  | SALESMAN  | 1600 |         1760 |
| BLAKE  | MANAGER   | 2850 |         3135 |
| CLARK  | MANAGER   | 2450 |         2695 |
```

Si vous jouez avec des additions ou des concaténations de chaînes, vous allez rapidement vous heurter aux valeurs **NULL**. **NULL** est une valeur particulière qui correspond à une valeur non remplie et qui n'a rien à voir avec la valeur entière nulle qu'est **0**.

Dans une expression, si un opérande est **NULL** alors le résultat de toute l'expression est **NULL**. Pour remplacer les valeurs **NULL** par une valeur neutre telle que zéro ou une chaîne vide, on pourra utiliser la fonction **`ifnull(champ,valeurremplacement)`** ou son équivalent de la norme SQL ANSI **`coalesce`**.

Fichier

```
SELECT ename,sal,comm,sal+comm PasBon,sal+ifnull(comm,0) SALCOMM
FROM emp
```

Ici, nous réalisons l'addition entre le salaire et les commissions qui sont souvent à **NULL**.

Terminal

```
+--------+------+------+--------+---------+
| ename  | sal  | comm | PasBon | SALCOMM |
+--------+------+------+--------+---------+
| ADAMS  | 1100 | NULL |   NULL |    1100 |
| ALLEN  | 1600 |  300 |   1900 |    1900 |
| BLAKE  | 2850 | NULL |   NULL |    2850 |
```

On voit que ceux qui n'ont pas de commissions ont **NULL** dans la colonne **PasBon** qui est une simple addition alors que la colonne **SALCOMM** est toujours correctement calculée.

Nous pouvons aussi faire des calculs conditionnels, comme, par exemple, augmenter de 10% pour le département **20** de 15% pour le département **30** et tel quel pour le reste. Nous utiliserons ici l'opérateur **case** qui permet d'exprimer des conditions sur des égalités ou sur des expressions complexes.

Fichier

```
SELECT ename,deptno,sal,
  round(CASE deptno
           when 20 then 1.1
           when 30 then 1.15
           else 1 END * sal) as NEW_SAL
FROM emp
```

Nous obtiendrons le résultat suivant :

Terminal

```
+--------+--------+------+---------+
| ename  | deptno | sal  | NEW_SAL |
+--------+--------+------+---------+
| ADAMS  |     20 | 1100 |    1210 |
| ALLEN  |     30 | 1600 |    1840 |
| BLAKE  |     30 | 2850 |    3278 |
| CLARK  |     10 | 2450 |    2450 |
```

Vous remarquez, que **case** est un opérateur comme un autre, qui permet d'écrire une expression et qu'il est combinable avec une multiplication et même mis en paramètre d'une fonction.

L'opérateur **case** a une variante syntaxique qui permet d'écrire des conditions dans la clause **when** et n'est plus relative à une variable spécifiée derrière le mot-clé **case**. Au passage, je montre ci-dessous qu'il est possible de mettre une expression dans le **then**, en disant que les employés des départements **>=30** sont augmentés de **100** via une addition.

Fichier

```
SELECT ename,deptno,sal,
  round(CASE
           when deptno=20 then sal*1.1
           when deptno>=30 then sal+100
           else sal END ) as NEW_SAL
FROM emp
```

Tous les SGBDR intègrent de nombreuses fonctions pour faire des calculs, elles ne sont quasiment pas standardisées, je vous invite à étudier celles qui sont disponibles sur votre SGBDR.

2. LES JOINTURES

La force des bases de données relationnelles est de répartir les données dans de multiples tables et d'être capable de lier ces tables efficacement au moyen de jointures.

Ici une jointure qui nous affiche les employés travaillant à **DALLAS** :

Fichier

```
SELECT e.ename,d.dname
FROM emp e,dept d
WHERE e.DEPTNO=d.DEPTNO and d.loc='DALLAS'
```

Nous obtiendrons le résultat suivant :

Terminal

```
+-------+----------+
| ename | dname    |
+-------+----------+
| ADAMS | RESEARCH |
| FORD  | RESEARCH |
| JONES | RESEARCH |
| SCOTT | RESEARCH |
| SMITH | RESEARCH |
+-------+----------+
```

Attention à divers points, toutes les colonnes qui sont présentes dans plus d'une table doivent impérativement être préfixées par l'alias de la table (ici **e** et **d**).

Avec **MySQL/MariaDB**, sur des OS sensibles à la casse (comme **Linux**) les alias des tables et les noms des tables sont *case-sensitive*, ce qui peut causer des désagréments en cas de développement réalisé sous **Windows** (qui n'est pas sensible à la casse) et finalement hébergé sur un serveur Linux.

Ici, nous avons utilisé la notation historique des jointures en séparant les noms des tables dans la clause **from** avec des virgules et en mélangeant dans la clause **where** les clauses de jointures et les prédicats de filtrage.

Je vous conseille de plutôt utiliser la notation ANSI qui est maintenant disponible dans la grande majorité des SGBDR qui nous donnera cette requête :

Fichier

```
SELECT e.ename,d.dname
FROM emp e
JOIN dept d on e.DEPTNO=d.DEPTNO
WHERE d.loc='DALLAS'
```

Un effet pas toujours désiré des jointures, est, que si on n'arrive pas à faire la jointure, la donnée non jointe disparaît. Par exemple, si j'affiche les employés avec leur manager, je vais avoir un problème avec **KING** qui n'a pas de manager.

Afin de traiter ce problème, il est possible de faire des jointures ouvertes à droite ou à gauche qui permettent de dire que si on n'arrive pas à se joindre avec l'autre table, on affiche quand même les données de la table de droite ou de gauche :

Fichier

```
SELECT e.ename employe,m.ename as manager
FROM emp e
LEFT OUTER JOIN emp m on e.mgr=m.empno
WHERE e.deptno=10
```

Dans cette requête, nous affichons quand même **KING** dans la table de gauche (**left**) :

Terminal

```
+---------+---------+
| employe | manager |
+---------+---------+
| CLARK   | KING    |
| KING    | NULL    |
| MILLER  | CLARK   |
+---------+---------+
```

Sans la clause **left outer** nous n'aurions eu que deux lignes (**clark** et **miller**). Le mot clé **outer** est optionnel ; c'est **left** qui fait tout le travail et on peut écrire **LEFT JOIN**.

Ce mécanisme peut aussi être utilisé pour faire ce que nous appelons une anti-jointure :

Fichier

```
SELECT e.ename employe,m.ename as manager
FROM emp e
RIGHT OUTER JOIN emp m on e.mgr=m.empno
WHERE e.empno is null
```

Dans cette requête, nous affichons les employés avec leur manager, en spécifiant que nous voulons aussi les lignes de la table de droite (les managers) même s'il n'y a pas de correspondance. Puis nous mettons une condition sur la table de gauche afin de ne conserver que ces lignes-là et supprimer les lignes correctement jointes.

Terminal

```
+---------+---------+
| employe | manager |
+---------+---------+
| NULL    | ADAMS   |
| NULL    | ALLEN   |
| NULL    | JAMES   |
| NULL    | MARTIN  |
| NULL    | MILLER  |
| NULL    | SMITH   |
| NULL    | TURNER  |
| NULL    | WARD    |
+---------+---------+
```

Nous obtenons du coup, les managers de personne (la colonne **employe** est laissée seulement à des fins d'explication et n'a pas à être affichée).

Ça peut paraître peu intuitif au premier abord, mais c'est finalement très logique et assez efficace.

On peut tout à fait le faire avec un **left outer join**, il suffit de permuter les deux tables **e** et **m**.

3. LES SOUS-REQUÊTES

Le SQL permet d'utiliser une requête comme ensemble de données au sein d'une requête SQL au travers de sous-requêtes. Ces sous-requêtes peuvent être présentes à plusieurs endroits de la requête principale.

3.1 Conditions avec des sous-requêtes

La plus ancienne forme de sous-requête est dans la clause **where**, typiquement avec l'utilisation de l'opérateur **IN**, mais il est aussi possible de les utiliser avec un **=** ou un **>** :

Fichier

```
SELECT empno,ename FROM emp
WHERE empno in (SELECT mgr FROM emp );
```

Ici, nous récupérons la liste des employés qui ont leur numéro d'employé présent dans la colonne **mgr**. Dit autrement, nous obtenons la liste de ceux qui sont managers d'un employé.

Terminal

```
+-------+-------+
| empno | ename |
+-------+-------+
|  7902 | FORD  |
|  7698 | BLAKE |
|  7839 | KING  |
|  7566 | JONES |
|  7788 | SCOTT |
|  7782 | CLARK |
+-------+-------+
```

Nous pouvons demander l'inverse avec **NOT IN**, mais là il y a un piège qui apparaît (sur la majorité des SGBDR du marché) : la sous-requête ne doit pas retourner de valeurs **NULL**. Nous devons donc les exclure de la sous-requête ainsi :

Fichier

```
SELECT empno,ename from emp
WHERE empno not in (SELECT mgr FROM emp WHERE mgr is not null);
```

Le résultat est :

Terminal

```
+-------+--------+
| empno | ename  |
+-------+--------+
|  7369 | SMITH  |
|  7499 | ALLEN  |
|  7521 | WARD   |
|  7654 | MARTIN |
|  7844 | TURNER |
|  7876 | ADAMS  |
|  7900 | JAMES  |
|  7934 | MILLER |
+-------+--------+
```

Dans cette approche, la sous-requête peut être considérée comme une table temporaire créée avant l'exécution de la requête principale. Une autre approche est possible avec les sous-requêtes corrélées (ou synchronisées). Dans cette approche, la sous-requête va être exécutée pour chaque ligne de la requête principale.

Fichier

```
SELECT empno,ename FROM emp e
WHERE not exists (SELECT 1 FROM emp s WHERE s.mgr=e.empno);
```

Cette requête nous donne le même résultat que la requête précédente, mais avec une logique assez différente. Nous utilisons ici le prédicat unaire (il n'y a rien à sa gauche) **not exists** qui teste la présence de lignes dans la sous-requête. S'il n'y en a pas, il est vrai. Cet opérateur n'a de sens qu'avec des sous-requêtes corrélées, car puisqu'il n'y a pas de partie gauche à la condition, il faut bien que la partie droite varie pour que ça ne soit pas un prédicat constant.

Vous remarquez la référence à **e.empno** dans la sous-requête. C'est cette référence qui provoque le changement de logique. Du coup pour chaque ligne de **e** nous allons

récupérer la liste des subordonnés de **e**. La seule information utile dans cette liste est de savoir si elle est vide ou pas, d'où la présence dans la clause **select** du littéral constant **1**. Nous aurions pu mettre n'importe quel champ ou ***** vu que les valeurs ne sont pas utilisées, mais je trouve plus clair d'utiliser une constante bien que contrairement au passé, il n'y ait plus d'écart de performance entre mettre ***** ou **1** ou **'X'**.

3.2 Tables sous-requête

La seconde forme de sous-requête est l'utilisation dans la clause **from**. Ici les sous-requêtes doivent être indépendantes, et tout comme des tables, devront être liées par des clauses de jointure. Ce type de sous-requête est très pratique dans de nombreuses situations où on se retrouve à vouloir exécuter une requête sur le résultat d'une requête.

Fichier

```
SELECT *
FROM (SELECT empno, ename,sal, comm, sal+ifnull(comm,0) salcomm
 FROM emp ) e
WHERE salcomm>=3000
```

Dans cet exemple, cette solution permet de pouvoir utiliser la colonne **salcomm** dans la clause **where** ce qui, sur une requête simple, implique de devoir réécrire l'expression dans la clause **where** ainsi **sal+ifnull(comm,0)>=3000** en plus de l'écrire dans la clause **select** pour l'afficher.

Terminal

```
+-------+-------+------+------+---------+
| empno | ename | sal  | comm | salcomm |
+-------+-------+------+------+---------+
|  7788 | SCOTT | 3000 | NULL |    3000 |
|  7839 | KING  | 5000 | NULL |    5000 |
|  7902 | FORD  | 3000 | NULL |    3000 |
+-------+-------+------+------+---------+
```

Dans cet autre exemple, nous utilisons notre sous-requête avec une autre table ; nous devons donc faire une jointure entre les employés et notre sous-requête qui donne le salaire moyen par job, comme nous l'aurions fait entre deux tables.

Fichier

```
SELECT ename, e.job, sal,saljob
  ,round((sal/saljob-1)*100) PctEcart
FROM emp e
JOIN (SELECT job,avg(sal) saljob
      FROM emp group by job ) m on e.job=m.job
```

Ceci nous permet d'afficher simultanément le salaire d'un employé et le salaire moyen des employés ayant le même job que lui et faire un calcul dessus pour calculer le % d'écart.

Terminal

```
+--------+-----------+------+-----------+----------+
| ename  | job       | sal  | saljob    | PctEcart |
+--------+-----------+------+-----------+----------+
| SMITH  | CLERK     |  800 | 1037.5000 |      -23 |
| ALLEN  | SALESMAN  | 1600 | 1400.0000 |       14 |
| WARD   | SALESMAN  | 1250 | 1400.0000 |      -11 |
| JONES  | MANAGER   | 2975 | 2758.3333 |        8 |
| MARTIN | SALESMAN  | 1250 | 1400.0000 |      -11 |
| BLAKE  | MANAGER   | 2850 | 2758.3333 |        3 |
```

```
| CLARK   | MANAGER   | 2450 | 2758.3333 |      -11 |
| SCOTT   | ANALYST   | 3000 | 3000.0000 |        0 |
| KING    | PRESIDENT | 5000 | 5000.0000 |        0 |
| TURNER  | SALESMAN  | 1500 | 1400.0000 |        7 |
| ADAMS   | CLERK     | 1100 | 1037.5000 |        6 |
| JAMES   | CLERK     |  950 | 1037.5000 |       -8 |
| FORD    | ANALYST   | 3000 | 3000.0000 |        0 |
| MILLER  | CLERK     | 1300 | 1037.5000 |       25 |
+--------+-----------+------+-----------+----------+
```

3.3 Expression sous-requête

Le troisième cas d'usage est l'utilisation d'une sous-requête dans la clause **select** ; on parle alors d'expression sous-requête. Dans ce cas la requête ne doit ramener qu'une seule ligne et une seule colonne :

Fichier

```
SELECT d.*,(SELECT count(*) FROM emp e WHERE e.deptno=d.deptno) NbrEmp
FROM dept d;
```

Le résultat est le suivant :

Terminal

```
+--------+------------+----------+--------+
| DEPTNO | DNAME      | LOC      | NbrEmp |
+--------+------------+----------+--------+
|     10 | ACCOUNTING | NEW YORK |      3 |
|     20 | RESEARCH   | DALLAS   |      5 |
|     30 | SALES      | CHICAGO  |      6 |
|     40 | OPERATIONS | BOSTON   |      0 |
+--------+------------+----------+--------+
```

Cette technique requiert souvent l'utilisation de sous-requêtes corrélées et n'est pas toujours très performante, mais dans certaines situations peut être une solution parfaitement adaptée.

Nous aurions pu essayer de répondre à la même question avec cette requête :

Fichier

```
SELECT e.deptno,dname,loc,count(*)  Nbr
FROM emp e
JOIN dept d ON e.deptno=d.deptno
GROUP BY e.deptno,dname,loc
```

Nous obtenons :

Terminal

```
+--------+------------+----------+-----+
| deptno | dname      | loc      | Nbr |
+--------+------------+----------+-----+
|     10 | ACCOUNTING | NEW YORK |   3 |
|     20 | RESEARCH   | DALLAS   |   5 |
|     30 | SALES      | CHICAGO  |   6 |
+--------+------------+----------+-----+
```

Mais nous voyons que le résultat obtenu n'est pas exactement le même, puisque le département **40**, qui n'a pas d'employés, n'apparaît pas pour dire qu'il n'a pas d'employés. Suivant ce que l'on cherche à savoir, ce n'est pas forcément gênant.

Si on souhaite obtenir le même résultat que la première requête avec d'autres techniques, voilà deux autres solutions :

Fichier

```
SELECT d.deptno,dname,loc,count(e.deptno)  Nbr
FROM emp e
RIGHT JOIN dept d on e.deptno=d.deptno
GROUP BY d.deptno,dname,loc
```

Ici on fait une jointure ouverte sur la table **dept** qui permet de faire apparaître tous les départements et on transforme le **count(*)** en **count** sur un champ normalement rempli de la table **emp** (la clé primaire est parfaitement indiquée dans ce cas là).

Une autre solution est d'utiliser une sous-requête dans le **from** ainsi :

Fichier

```
SELECT d.deptno,dname,loc,coalesce(Nbr,0) Nbr
FROM dept d
LEFT JOIN (SELECT deptno,count(*) Nbr
  FROM emp
  GROUP BY deptno ) e
ON e.deptno=d.deptno
```

Ici l'idée est de calculer le nombre d'employés par département à partir de la table **emp**, de faire une jointure ouverte entre ce résultat et la table **dept**, puis de dire que si on est sur une ligne non jointe avec la sous-requête **e**, donc que **Nbr** est **NULL**, alors on remplace **NULL** par **0**.

4. CALCULS D'AGRÉGATION

Les calculs d'agrégation (que nous avons déjà aperçus dans le paragraphe précédent) sont des calculs qui vont nous permettre de faire des calculs sur un ensemble de lignes et d'avoir moins de lignes à la fin. Ils peuvent être simples et dans ce cas ils donneront une seule ligne :

Fichier

```
SELECT sum(sal) SalManager
FROM emp
WHERE job='MANAGER';
```

Résultat :

Terminal

```
+------------+
| SalManager |
+------------+
|       8275 |
+------------+
```

On peut aussi utiliser des groupes et dans ce cas ils retourneront une ligne par groupe :

Fichier

```
SELECT job,sum(sal) SalJob
FROM emp
GROUP BY job;
```

Le résultat obtenu est :

Terminal

```
+-----------+--------+
| job       | SalJob |
+-----------+--------+
| ANALYST   |   6000 |
| CLERK     |   4150 |
| MANAGER   |   8275 |
| PRESIDENT |   5000 |
| SALESMAN  |   5600 |
+-----------+--------+
```

De façon un peu moins classique, on peut baser les groupes sur des expressions. Par exemple, ici des tranches de 1000€ :

Fichier

```
SELECT concat(truncate(sal,-3),'-',truncate(sal,-3)+999) Tranche,count(*) Nbr
FROM emp
GROUP BY truncate(sal,-3);
```

Nous obtenons :

Terminal

```
+-----------+-----+
| Tranche   | Nbr |
+-----------+-----+
| 0-999     |   2 |
| 1000-1999 |   6 |
| 2000-2999 |   3 |
| 3000-3999 |   2 |
| 5000-5999 |   1 |
+-----------+-----+
```

On remarque qu'on répète un peu l'expression **`truncate(sal,-3)`**. On pourra éviter ceci en utilisant une sous-requête dans la clause **`from`** :

Fichier

```
SELECT concat(Tranche,'-',Tranche+999) Intervalle,count(*) Nbr
FROM (SELECT truncate(sal,-3) Tranche,e.*
 FROM emp e ) EI
GROUP BY Tranche;
```

Ça sera d'autant plus pertinent si on utilise une expression plus complexe, par exemple :

Fichier

```
SELECT cadre,sum(sal) SommeSal, count(*) Nbr
FROM ( SELECT case when job in ('PRESIDENT','MANAGER') then 'Y' else 'N'
end cadre, e.*
   FROM emp e) EC
GROUP BY Cadre;
```

Ceci nous donnera la masse salariale et le nombre d'employés qui sont cadres ou pas.

Terminal

```
+-------+----------+-----+
| cadre | SommeSal | Nbr |
+-------+----------+-----+
| N     |    15750 |  10 |
| Y     |    13275 |   4 |
+-------+----------+-----+
```

On pourrait souhaiter avoir cette information par département en ajoutant **`deptno`** dans la clause **`group by`** :

Fichier

```
SELECT deptno,cadre,sum(sal) SommeSal, count(*) Nbr
FROM ( SELECT case when job in ('PRESIDENT','MANAGER') then 'Y' else 'N'
end cadre, e.*
  FROM emp e) EC
GROUP BY deptno, Cadre;
```

Cette requête renvoie :

Terminal

```
+--------+-------+----------+-----+
| DEPTNO | cadre | SommeSal | Nbr |
+--------+-------+----------+-----+
|     10 | N     |     1300 |   1 |
|     10 | Y     |     7450 |   2 |
|     20 | N     |     7900 |   4 |
|     20 | Y     |     2975 |   1 |
|     30 | N     |     6550 |   5 |
|     30 | Y     |     2850 |   1 |
+--------+-------+----------+-----+
```

Nous avons l'information, mais ça n'est pas forcement super lisible. Quand on a une liste connue et de taille fixe de catégories, il peut être intéressant de les présenter comme ceci :

Terminal

```
+--------+---------------+----------+------------------+-------------+
| DEPTNO | SommeSalCadre | NbrCadre | SommeSalNonCadre | NbrNonCadre |
+--------+---------------+----------+------------------+-------------+
|     10 |          7450 |        2 |             1300 |           1 |
|     20 |          2975 |        1 |             7900 |           4 |
|     30 |          2850 |        1 |             6550 |           5 |
+--------+---------------+----------+------------------+-------------+
```

Ceci est fait avec la requête suivante :

Fichier

```
SELECT deptno
      ,sum(case when cadre='Y' then sal end) SommeSalCadre
      ,count(case when cadre='Y' then 1 end) NbrCadre
      ,sum(case when cadre='N' then sal end) SommeSalNonCadre
      ,count(case when cadre='N' then 1 end) NbrNonCadre
FROM ( SELECT case when job in ('PRESIDENT','MANAGER') then 'Y' else 'N' end
cadre, e.*
       FROM emp e) EC
GROUP BY deptno;
```

Cette requête fait des calculs d'agrégats conditionnels et s'appuie sur le fait que les fonctions d'agrégations ignorent les valeurs **NULL**, ici provoquées par l'absence de **else** dans les expressions **case**.

À présent, revenons sur la requête du nombre d'employés par tranche, vous avez probablement remarqué qu'il n'y avait pas de tranche 4000-4999. Cela vient du fait qu'il n'y avait pas d'employés dans cette tranche pour la générer or, il se pourrait qu'on vous demande de faire une requête dans laquelle cette tranche apparaîtrait avec la valeur **0** et là vous allez être bien embêté en SQL (alors que vous ne le seriez pas trop en **PHP**). Pour faire ça en SQL, il nous faut un générateur séquentiel qui va nous permettre de générer toutes les lignes que nous voudrions avoir. Certains SGBDR ont des fonctions prévues pour ça (**generate_series** sous **PostgreSQL** et les CTE pour **MS SQL** & Oracle). Mais MySQL non (cependant c'est pour bientôt, cf. section sur le CTE). Sous MySQL, on pourra s'en sortir en listant une table qui a plus de lignes que la séquence que l'on veut générer et afficher le numéro de ligne.

La requête ci-dessous nous affiche 10 lignes allant de **1** à **10** en utilisant la table **emp** qui en a 14.

Fichier

```
SELECT @rownum:=@rownum+1 rownum from emp t, (SELECT @rownum:=0) r limit 10;
```

Nous allons utiliser cette technique que nous allons adapter pour générer une séquence allant de **0** à **5000** avec un pas de **1000** avec laquelle nous allons faire une jointure externe pour résoudre notre problème :

Fichier

```
SELECT concat(seq.Tranche,'-',seq.Tranche+999) Intervalle,count(ei.empno) Nbr
FROM (select @rownum:=@rownum+1000 tranche from emp t, (SELECT @rownum:=-1000)
r limit 6) seq
LEFT JOIN (SELECT truncate(sal,-3) Tranche,e.*
 FROM emp e ) ei ON ei.tranche=seq.tranche
GROUP BY seq.Tranche;
```

On peut voir ci-dessous que ça marche :

Terminal

```
+------------+-----+
| Intervalle | Nbr |
+------------+-----+
| 0-999      |   2 |
| 1000-1999  |   6 |
| 2000-2999  |   3 |
| 3000-3999  |   2 |
| 4000-4999  |   0 |
| 5000-5999  |   1 |
+------------+-----+
```

La colonne intervalle est à présent fabriquée à partir de la sous-requête **seq** et nous comptons le nombre d'employés (**count(ei.empno)** à la place de **count(*)**, sinon on compterait aussi les lignes non jointes et on aurait la valeur **1** sur la ligne de la tranche **4000** au lieu de **0**).

5. FONCTIONS ANALYTIQUES

La norme SQL intègre de façon relativement récente (SQL:2003) des fonctions analytiques. MySQL ne les intègre pas, mais MariaDB les intègre dans sa version 10.2.0 (bêta au moment de l'écriture de cet article).

5.1 Fonctions de classement

Voilà un exemple de fonctions disponibles :

Fichier

```
SELECT ename,sal
 ,rank() over(order by sal desc) rang
 ,dense_rank() over(order by sal desc) dense
 ,percent_rank() over(order by sal desc) centile
 ,ntile(4) over(order by sal desc) quartile
 ,row_number() over(order by sal desc) no_ligne
 ,rank() over(order by hiredate ) anci
FROM emp
ORDER BY sal desc
```

Ce qui nous donne ceci :

Terminal

```
+--------+------+------+-------+--------------+----------+----------+------+
| ename  | sal  | rang | dense | centile      | quartile | no_ligne | anci |
+--------+------+------+-------+--------------+----------+----------+------+
| KING   | 5000 |    1 |     1 | 0.0000000000 |        1 |        1 |    9 |
| SCOTT  | 3000 |    2 |     2 | 0.0769230769 |        1 |        2 |   13 |
| FORD   | 3000 |    2 |     2 | 0.0769230769 |        1 |        3 |   10 |
| JONES  | 2975 |    4 |     3 | 0.2307692308 |        1 |        4 |    4 |
| BLAKE  | 2850 |    5 |     4 | 0.3076923077 |        2 |        5 |    5 |
| CLARK  | 2450 |    6 |     5 | 0.3846153846 |        2 |        6 |    6 |
| ALLEN  | 1600 |    7 |     6 | 0.4615384615 |        2 |        7 |    2 |
| TURNER | 1500 |    8 |     7 | 0.5384615385 |        2 |        8 |    7 |
| MILLER | 1300 |    9 |     8 | 0.6153846154 |        3 |        9 |   12 |
| MARTIN | 1250 |   10 |     9 | 0.6923076923 |        3 |       10 |    8 |
| WARD   | 1250 |   10 |     9 | 0.6923076923 |        3 |       11 |    3 |
| ADAMS  | 1100 |   12 |    10 | 0.8461538462 |        4 |       12 |   14 |
| JAMES  |  950 |   13 |    11 | 0.9230769231 |        4 |       13 |   10 |
| SMITH  |  800 |   14 |    12 | 1.0000000000 |        4 |       14 |    1 |
+--------+------+------+-------+--------------+----------+----------+------+
```

Ici, nous effectuons principalement des analyses suivant un seul axe qui est le classement des salaires par ordre décroissant. Il est spécifié pour chacune des expressions dans la clause **over** associée à chaque fonction ainsi : **over(order by sal desc)** sauf pour la colonne **anci** qui affiche le classement suivant l'ancienneté.

Voici la description des fonctions employées :

Fonction	Description
rank()	Affiche le classement
dense_rank()	Affiche le classement, mais en cas d'ex æquo le classement suivant n'est pas sauté. Ex : Il y a deux seconds et un troisième alors qu'avec **rank()** il y a deux seconds et un quatrième.
row_number()	Affiche le numéro de ligne, similaire à **rank()**, mais en ignorant les ex æquo qui auront deux numéros distincts.
NTile(N)	Ici utilisé avec N=4, donne le quartile, c'est-à-dire qu'on répartit les individus en 4 ensembles égaux (aux problèmes de multiple de 4 près) et on dit dans quel ensemble ils sont. Si N = 10 on parle de décile, si N = 100 on parle de centile.
percent_rank()	Donne le centile dans lequel on se situe en répartissant les valeurs entre 0 et 1.

La clause **over** permet de spécifier l'ordre pour les classements, mais aussi, de façon optionnelle, une notion de partition qui remet les compteurs à zéro quand on change de partition. Ci-dessous, un exemple qui affiche le classement des salaires par département :

Fichier

```
SELECT deptno,ename,sal
 ,rank() over(partition by deptno ORDER BY sal desc) rang
FROM emp
ORDER BY deptno,sal desc
```

Le résultat est :

Terminal

```
+--------+--------+------+------+
| deptno | ename  | sal  | rang |
+--------+--------+------+------+
|     10 | KING   | 5000 |    1 |
|     10 | CLARK  | 2450 |    2 |
```

```
|     10 | MILLER | 1300 |    3 |
|     20 | FORD   | 3000 |    1 |
|     20 | SCOTT  | 3000 |    1 |
|     20 | JONES  | 2975 |    3 |
|     20 | ADAMS  | 1100 |    4 |
|     20 | SMITH  |  800 |    5 |
|     30 | BLAKE  | 2850 |    1 |
|     30 | ALLEN  | 1600 |    2 |
|     30 | TURNER | 1500 |    3 |
|     30 | WARD   | 1250 |    4 |
|     30 | MARTIN | 1250 |    4 |
|     30 | JAMES  |  950 |    6 |
+--------+--------+------+------+
```

Ces fonctions peuvent être utiles pour sortir des Top N comme les N meilleurs, éventuellement par catégorie.

Cela se fera en mettant une condition sur le **`rank()`** dans la clause **`where`**, mais cela est interdit (spécifié dans la norme) et il faudra donc passer par une sous-requête dans le **`from`** ainsi :

Fichier

```
SELECT deptno,ename,sal
FROM (SELECT deptno,ename,sal
            ,rank() over(partition by deptno ORDER BY sal desc) rang
            FROM emp ) e
WHERE rang=1
ORDER BY deptno
```

Résultat :

Terminal

```
+--------+-------+------+
| deptno | ename | sal  |
+--------+-------+------+
|     10 | KING  | 5000 |
|     20 | SCOTT | 3000 |
|     20 | FORD  | 3000 |
|     30 | BLAKE | 2850 |
+--------+-------+------+
```

5.2 Fonctions de fenêtrage

Il est possible d'utiliser les fonctions d'agrégats avec des fonctions de fenêtrages. Ainsi la fonction s'appliquera aux lignes autour, par exemple si on souhaite étudier l'évolution des salaires en fonctions des dates d'embauches :

Fichier

```
SELECT hiredate,sal
  , avg(sal) over(ORDER BY hiredate ROWS BETWEEN 3 preceding AND current
row) MoyenneGlissante3
 , avg(sal) over(ORDER BY hiredate ROWS BETWEEN unbounded preceding AND
current row) MoyenneGlissante
  , avg(sal) over(ORDER BY hiredate ) MoyGliSimple
FROM emp
ORDER BY hiredate
```

Nous obtenons :

Terminal

```
+------------+------+--------------------+------------------+--------------+
| hiredate   | sal  | MoyenneGlissante3  | MoyenneGlissante | MoyGliSimple |
+------------+------+--------------------+------------------+--------------+
| 1980-12-17 |  800 |           800.0000 |         800.0000 |     800.0000 |
| 1981-02-20 | 1600 |          1200.0000 |        1200.0000 |    1200.0000 |
| 1981-02-22 | 1250 |          1216.6667 |        1216.6667 |    1216.6667 |
| 1981-04-02 | 2975 |          1656.2500 |        1656.2500 |    1656.2500 |
| 1981-05-01 | 2850 |          2168.7500 |        1895.0000 |    1895.0000 |
| 1981-06-09 | 2450 |          2381.2500 |        1987.5000 |    1987.5000 |
| 1981-09-08 | 1500 |          2443.7500 |        1917.8571 |    1917.8571 |
| 1981-09-28 | 1250 |          2012.5000 |        1834.3750 |    1834.3750 |
| 1981-11-17 | 5000 |          2550.0000 |        2186.1111 |    2186.1111 |
| 1981-12-03 | 3000 |          2687.5000 |        2267.5000 |    2147.7273 |
| 1981-12-03 |  950 |          2550.0000 |        2147.7273 |    2147.7273 |
| 1982-01-23 | 1300 |          2562.5000 |        2077.0833 |    2077.0833 |
| 1987-04-19 | 3000 |          2062.5000 |        2148.0769 |    2148.0769 |
| 1987-05-23 | 1100 |          1587.5000 |        2073.2143 |    2073.2143 |
+------------+------+--------------------+------------------+--------------+
```

MoyenneGlissante3 affiche la moyenne glissante des quatre dernières lignes en incluant la ligne courante. **MoyenneGlissante** affiche la moyenne glissante à partir de la première ligne (**unbounded preceding**). **MoyGliSimple** donne le même résultat, c'est le comportement par défaut quand on ne spécifie pas de fenêtre avec juste une clause **over()**.

La norme intègre d'autres fonctions qui ne sont pas encore couvertes par MariaDB, mais l'essentiel est là. Pour celles déjà implémentées par MariaDB, je ne doute pas que MySQL ne manquera pas de rattraper l'avance prise par MariaDB sur ce sujet-là.

6. CTE

Les CTE (*Common Table Expression*) quant à elles font partie de la norme SQL:1999, mais apparaissent à peine dans MariaDB (Bêta 10.2.1 et 10.2.2 pour les récursives) et MySQL (Lab Release 8.0.0).

Les CTE sont un mécanisme comparable aux sous-requêtes dans le **from** hormis qu'elles sont réutilisables dans plusieurs sous-requêtes comme le serait une table.

Si nous reprenons une requête précédemment écrite, la transformation de la sous-requête en CTE nous donne ceci :

Fichier

```
WITH EmpByDept as (SELECT deptno,count(*) Nbr
  from emp
  group by deptno )
SELECT d.deptno,dname,loc,coalesce(Nbr,0) Nbr
FROM dept d
LEFT JOIN  EmpByDept e on e.deptno=d.deptno;
```

Et nous obtenons alors :

Terminal

```
+--------+------------+----------+------+
| deptno | dname      | loc      | Nbr  |
+--------+------------+----------+------+
|     10 | ACCOUNTING | NEW YORK |    3 |
|     20 | RESEARCH   | DALLAS   |    5 |
|     30 | SALES      | CHICAGO  |    6 |
|     40 | OPERATIONS | BOSTON   |    0 |
+--------+------------+----------+------+
```

On voit ici que **EmpByDept** n'est pas un alias, mais est considéré comme une table.

Attention, le fait qu'une requête **select** ne commence pas par le mot-clé **select** peut poser des soucis dans une multitude d'outils et de bibliothèques (cette remarque est vraie pour tous les SGBDR qui implémentent les CTE). Cependant, le principal intérêt des CTE n'est pas ici, mais est dans les requêtes récursives.

La syntaxe peut légèrement varier suivant les SGBDR:

Fichier

```
WITH RECURSIVE H(empno,mgr,ename,niveau,ancetre,ancetres)
as (
SELECT empno,mgr,ename
,1 niveau,CAST('' as CHAR(20))  ancetre
,CAST('' as CHAR(50))  ancetres
FROM emp
WHERE ename='JONES'
UNION ALL
SELECT E.empno,e.mgr,e.ename
   ,niveau+1 niveau
   ,h.ename ancetre
   ,concat(ancetres,',',h.ename) ancetres
FROM emp E join H on e.mgr=H.empno
)
SELECT * FROM H
```

Cet exemple nous affiche l'arbre hiérarchique en partant de **JONES** :

Terminal

```
 empno | mgr  | ename | niveau | ancetre |   ancetres
-------+------+-------+--------+---------+--------------
  7566 | 7839 | JONES |      1 |         |
  7902 | 7566 | FORD  |      2 | JONES   | ,JONES
  7788 | 7566 | SCOTT |      2 | JONES   | ,JONES
  7369 | 7902 | SMITH |      3 | FORD    | ,JONES,FORD
  7876 | 7788 | ADAMS |      3 | SCOTT   | ,JONES,SCOTT
```

Notre requête récursive est dans la clause **with** qui est en deux parties.

La première partie permet de ramener le jeu de données initial qui est dans le cas présent l'enregistrement de **JONES** (plus l'initialisation du niveau à **1** et il n'a pas d'ancêtres).

La seconde partie (après le **union all**) va nous ramener les lignes des récursions suivantes en faisant une jointure sur **H** (qui est la requête récursive) entre le manager (**H**) et le subordonné (**E**) et cela itérera tant que la seconde moitié retournera des lignes.

La seconde moitié peut faire référence à l'ancêtre lié et nous utilisons donc ceci pour gérer la colonne **niveau** qui est initialisée à **1** et augmente de **1** à chaque récursion et donne la profondeur courante du parcours de l'arbre.

La même astuce est utilisée pour générer les colonnes **ancetre** et **ancetres**. Vous noterez le **CAST('' as CHAR(20))** de la première requête qui permet de fixer la largeur de la colonne à 20 caractères, sinon elle serait de zéro et n'afficherait donc rien.

Dans cet autre exemple, nous utilisons le même principe pour fabriquer un générateur séquentiel, ici un exemple qui nous aurait servi pour un de nos exemples en générant des tranches de **0** à **5000** :

Fichier

```
with RECURSIVE Generateur(n)
 as (
  SELECT 0 n
  UNION ALL
  SELECT n+1000
  FROM Generateur
  WHERE n<5000
)
SELECT * FROM Generateur;
```

Résultat :

Terminal

```
  n
------
     0
 1000
 2000
 3000
 4000
 5000
```

Cette technique fonctionne aussi sous PostgreSQL mais on utilisera plutôt cette requête qui est plus simple et plus efficace (en espérant que MySQL s'en inspire un jour) :

Fichier

```
SELECT generate_series as n
FROM generate_series(0,5000,1000);
```

CONCLUSION

J'espère vous avoir démontré que le SQL pouvait vous rendre de grands services dans une écriture concise et relativement compréhensive à condition d'avoir consacré un minimum d'effort à l'apprendre.

Les traitements SQL seront généralement plus performants que les traitements sur les données brutes dans un autre langage (car il faut d'abord transférer ces données dans la mémoire de l'autre programme) en plus d'être plus simples à écrire (faire l'équivalent d'un **`group by`** en PHP est une hérésie malheureusement assez répandue : c'est lent et compliqué).

RÉFÉRENCE

[1] Les scripts pour recréer la base sous MySQL, Oracle et SQL Server sont disponibles ici : **http://www.altidev.com/livres.php** (en bas de la page, Base Employés (petite))

POUR ALLER PLUS LOIN

N'hésitez pas à lire les documentations des éditeurs, je vous promets qu'on peut y apprendre des choses dedans bien que la qualité soit inégale suivant les éditeurs.

Le livre de Christian Soutou : *« Programmer avec MySQL»*, Eyrolles, 2015. 5ème édition prévue début 2017. ▉

ÉTAPE 3

J'UTILISE LE LANGAGE SQL POUR ACCÉDER À MA BASE

AUGMENTEZ LES PERFORMANCES DE VOS BASES DE DONNÉES AVEC LES INDEX

Laurent NAVARRO

Les index sont un des éléments centraux de l'optimisation des bases de données, je vous propose d'étudier leur mise en œuvre.

Les index sont un mécanisme que l'on retrouve dans toutes les bases de données (et même ailleurs). Le concept d'index existe depuis plusieurs siècles (depuis l'antiquité en fait). L'index des SGBDR est inspiré de l'index que l'on trouve à la fin des livres, qui contient les mots importants d'un ouvrage et la page (ou les pages) à laquelle on va trouver ces mots.

La caractéristique d'un index est qu'il est organisé de façon à ce qu'il soit rapide de retrouver la clé recherchée à l'intérieur et qu'il nous fournisse une référence pour accéder rapidement à l'ensemble de l'enregistrement.

Je vous accorde que cette définition est quelque peu généraliste et ne vous a peut-être pas éclairée, nous allons donc l'illustrer avec l'exemple simple d'une liste d'employés qui sont identifiés par un numéro dans une table **MySQL** utilisant un moteur **MyISAM**. Pourquoi sur un moteur MyISAM, alors que ce dernier est considéré comme obsolète ? Tout simplement, parce que son moteur de stockage est basé sur des tables organisées en tas (*heap*) tout comme **Oracle**, **PostgreSQL** et **SQL Serveur** avec des tables *non-clustered* et que pour expliquer ce qu'est un index c'est le cas le plus simple. L'autre cas n'est pas fondamentalement différent.

1. INDEX SUR UNE TABLE EN TAS (HEAP)

Dans une table organisée en tas, les données sont posées plus ou moins dans l'ordre ou elles sont insérées dans le « fichier » contenant le tas (en vérité, c'est un peu plus compliqué que ça, mais dans cet article j'utiliserai volontairement une vision simplifiée, mais aussi fidèle que possible des choses).

Si vous écrivez la requête suivante :

Fichier

```
select * from emp where empno=1015
```

Le moteur va parcourir la totalité du tas afin d'extraire les lignes qui répondent à cette condition. Donc s'il y a 1 million d'enregistrements, il va tester 1 million de fois le prédicat **`empno=1015`**. Vous sentez probablement que ce n'est pas très optimal.

Est-ce que vous faites la même chose quand vous cherchez un mot dans votre dictionnaire papier ?

Bien sûr que non ! Pourquoi ? Parce que votre dictionnaire a une propriété particulière, les mots y sont classés par ordre alphabétique, ainsi vous êtes capable de trouver n'importe quels mots parmi les 100 000 que contient votre dictionnaire en quelques secondes et en ayant lu seulement 10 à 20 mots (soit 5 000 fois moins qu'une lecture séquentielle).

Un index de type B-Tree sur la colonne **`empno`** va contenir la liste des valeurs de façon ordonnée, ainsi, il va être possible de faire des recherches de façon efficace (en utilisant par exemple une recherche dichotomique). En plus des valeurs, l'index va contenir un pointeur vers l'enregistrement dans le tas nous permettant d'accéder à l'enregistrement complet. Ce mécanisme est illustré à la figure 1, page suivante.

Dans notre table **`emp`**, le champ **`empno`** est déclaré comme la clé primaire, ceci a pour effet de créer un index sur cette colonne à notre place, nous n'avons donc pas à déclarer d'index supplémentaire pour l'instant.

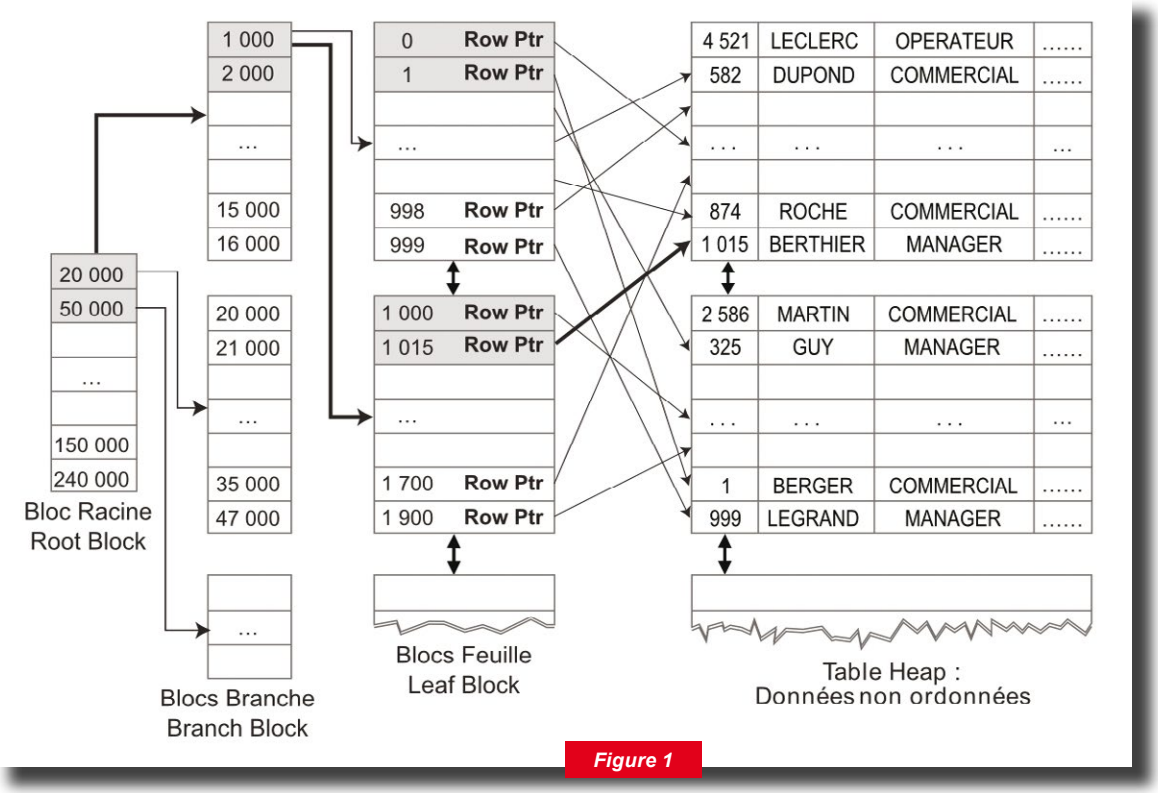


Figure 1

Index sur une table de type Heap.

Comment va être utilisé cet index ?

Lorsque la requête est soumise au SGBDR, celle-ci est analysée syntaxiquement et sémantiquement (lors de ces phases, des messages d'erreurs peuvent vous être remontés). Puis, le SGBDR va envisager les diverses façons possibles d'exécuter la requête. Dans le cas présent, il y en a deux :

A) Parcourir le tas de façon séquentielle ;

B) Utiliser l'index.

Pour chacune des solutions, l'optimiseur va estimer le coût de chacune des solutions et exécuter la moins coûteuse, qui dans le cas présent, sera en toute logique la solution B. Vous notez donc que dès qu'un index est présent, il n'y a rien à faire de particulier pour que le SGBDR l'utilise, il suffit qu'il considère que c'est la meilleure solution pour qu'il soit utilisé.

Détaillons un peu le fonctionnement de cet index.

À partir de la valeur présente dans le prédicat (**1015**), le SGBDR va rechercher dans le bloc racine quelle branche il doit utiliser, puis à partir de la branche il va localiser la feuille qui contient la valeur. Une fois trouvée la valeur, grâce au pointeur il va aller chercher dans le tas, par un accès direct, l'enregistrement (avec une table MyISAM, le pointeur est l'offset depuis le début de la table. Il occupe par défaut 4 octets ce qui limite la table à 4Gio, mais ceci est paramétrable pour les très grosses tables).

Les blocs « feuille » contiennent autant d'entrées qu'il y a de lignes dans la table. Les blocs « branche » contiennent les valeurs de début de chacun des blocs feuille, et le bloc racine contient les valeurs de début des blocs branche. S'il y a beaucoup de valeurs, il peut y avoir plusieurs niveaux de blocs branche.

La hauteur de l'arbre est variable, mais excède rarement trois ou quatre niveaux. Si on considère qu'un bloc non-feuille peut pointer sur 100 à 500 blocs (cela dépend de la taille de la clé), l'exemple précédent (d'une hauteur de trois) pourrait avoir 10 000 à 250 000 feuilles. La hauteur de l'arbre a une progression logarithmique par rapport au nombre d'enregistrements dans la table. Puisque le nombre d'accès nécessaires pour accéder à un élément dépend de la hauteur de l'arbre, il est donc lui aussi logarithmique.

Sans index, sur une table de **N** lignes, il faut en moyenne parcourir **N/2** lignes pour trouver une ligne unique et **N** lignes pour trouver toutes les occurrences d'une valeur dans une colonne non unique alors que, dans un index, il faut en moyenne parcourir une proportion de **log (N)** lignes pour trouver une ligne. Par exemple, pour une table d'un million d'enregistrements, sans index il faudra parcourir 1 000 000 d'enregistrements alors qu'avec un index il suffira de parcourir environ 30 enregistrements.

Étant donné la progression logarithmique, accéder à un enregistrement en passant par une clé indexée sur une table de 10 millions d'enregistrements ne nécessitera pas beaucoup plus d'accès que sur une table de 10 000 enregistrements.

Ceci implique aussi que pour des petites tables (<1000 lignes) les index auront un gain limité.

2. LES TABLES ORGANISÉES EN INDEX

Les IOT (*Index Organized Table*) sont des tables organisées en index, c'est le mode d'organisation des tables avec le moteur MySQL **InnoDB**, les tables *clustered* de MS SQL Serveur (format par défaut) et les tables IOT sous Oracle.

Dans une organisation IOT, les lignes sont stockées de façon ordonnée (suivant l'ordre de la clé primaire) et intégrées à un index. Pour résumer, les feuilles de l'index ne contiennent pas des pointeurs vers des enregistrements, mais les enregistrements eux-mêmes (ce qui implique accessoirement qu'il y a plus de feuilles que sur un index *heap*, car les feuilles contiennent moins d'entrées). Ceci est illustré dans la figure 2.

Figure 2

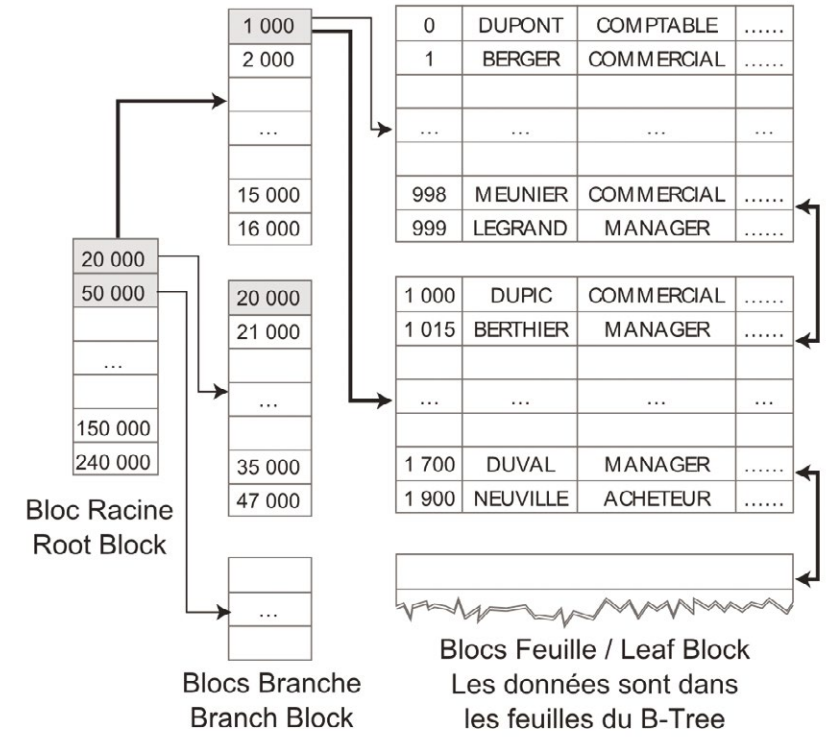


Table organisée en Index.

Avec InnoDB, s'il n'y a pas de clé primaire, une colonne invisible auto-incrémentée est créée et sert à indexer la table.

Ce type de table est plus efficace pour les recherches d'enregistrement suivant la clé primaire et les parcours d'intervalles suivant la clé primaire. Mais est moins performant quand les insertions ne sont pas faites suivant l'ordre de la clé primaire puisqu'il faut en permanence réorganiser la table.

3. LES INDEX SECONDAIRES

Nous avons pour l'instant évoqué les index à travers les clés primaires, mais il est possible de créer plusieurs index sur une table : on parle dans ce cas d'index secondaires.

Sur des tables organisées en tas (*heap*), il n'y a pas de différence entre un index de clé primaire et un index secondaire. Par contre sur des tables IOT, ils sont complètement différents. Sur une table IOT, l'index principal organise l'ordre des enregistrements dans la table, alors que les index secondaires, sont eux, similaires à des index secondaires des tables *heap*. Ils diffèrent un peu cependant, car, au lieu de contenir un pointeur direct vers la ligne, ils contiennent la clé primaire (sous Oracle, il y a en plus de la clé primaire, le RowID de la ligne au moment de son insertion, ceci permet, si elle n'a pas bougée, d'avoir un accès plus direct à l'enregistrement, on parle de RowID supposé). Sur ce type de table, il faut être vigilant sur la taille de la clé primaire, car cette dernière va être dupliquée autant de fois qu'il y aura d'index. Si elle est grosse, il peut être pertinent de mettre un champ auto-incrémenté en clé primaire technique et un index secondaire de type unique pour implémenter la clé primaire naturelle.

Pour créer un index, on utilisera la commande suivante :

Fichier
```
CREATE INDEX nom_index ON ma_table (ma_colonne)
```

Pour créer un index sur la colonne **ename** de la table **emp** on écrira donc :

Fichier
```
CREATE INDEX is_emp_ename ON emp(ename)
```

Sous MySQL, **CREATE INDEX** est un alias sur la commande **ALTER TABLE ADD INDEX**, on pourra donc aussi utiliser la commande suivante :

Fichier
```
ALTER TABLE emp ADD INDEX is_emp_ename(ename)
```

Nous avons précédemment évoqué le gain qu'apportait un index, mais quel est son coût ? Pour l'index associé à la clé primaire, la question peut paraître secondaire puisque nous n'avons pas vraiment le choix. Mais pour les index secondaires, il faut comparer le coût avec le bénéfice.

Un index a deux coûts. Il coûte de l'espace disque, qui est globalement constitué du contenu des feuilles de l'index (car il y a normalement beaucoup plus de feuilles que de branches). L'index contient une copie des valeurs des colonnes qui le constitue plus pour chaque ligne un pointeur vers la ligne. Le pointeur occupe par défaut 4 octets avec MyISAM et la taille de la clé primaire pour les tables InnoDB. Il faut ajouter à cet espace

l'espace vide laissé par défaut pour permettre d'insérer des lignes dans le désordre, ce qui représentera 10 à 20 % de la taille de l'index.

La somme de la taille des index peut être supérieure à la taille de la table elle-même.

Le second coût de l'index est le coût CPU de sa maintenance, puisqu'à chaque insertion, suppression d'enregistrement et modification de colonnes indexées, il faut mettre à jour les index associés. Ce coût n'est pas très important et peut généralement être négligé, mais s'il y a beaucoup d'index il peut ne plus être négligeable.

4. QUELLES COLONNES INDEXER ?

Nous avons vu comment se mettent en œuvre les index, mais maintenant se pose la question : quelle colonne faut-il indexer ?

Il faut indexer la clé primaire, mais le SGBDR s'en charge déjà pour vous puisqu'il y a toujours un index associé à la clé primaire, inutile donc de la réindexer.

Nous avons vu que l'optimiseur va calculer le coût de plusieurs chemins d'exécution pour une requête et choisir le moins coûteux. Il faut donc poser des index sur des colonnes qui vont permettre de réduire les coûts.

Le but est d'éviter de parcourir séquentiellement la totalité de la table si ce n'est pas nécessaire. Ce qui signifie qu'il va falloir a priori indexer les colonnes que l'on trouve généralement dans la clause **`where`** (et le **`on`** du **`join`**) de vos requêtes (à chaque fois que vous faites une requête sur une grosse table, demandez-vous s'il y a des index sur les clauses **`where`** de votre requête).

Les index sont utilisables sur des tests d'égalité et sur des intervalles de valeurs (opérateurs **`>`**,**`<`**), ils sont aussi utilisables sur les **`like 'Debut%'`** et sur les clauses de tri **`order by`**.

Ils ne sont par contre pas (ou moins bien) utilisés dans les cas suivants :

- utilisation de fonctions ou d'expressions sur la colonne indexée : **`abs (deptno)=10`** ;
- **`like`** avec un % au début : **`ename like '% A %'`** ;
- prédicat à faible sélectivité, c'est-à-dire si la proportion d'enregistrement filtré est élevée, par exemple **`sexe='M'`** sur la population d'un pays retournera approximativement la moitié de la table. Dans ce cas, il est plus efficace de parcourir toute la table que de faire des va-et-vient entre l'index et la table (cf.encadré sélectivité). De façon générale, au-delà de 5 % des enregistrements de la table ramenés par l'index, il ne sera pas utilisé (sauf si c'est juste pour un **`order by`**).

Il est fréquent d'indexer les champs utilisant une clé étrangère (**`foreign key`**), cette pratique est pertinente si la table référencée subit souvent des mises à jour de sa clé (car il faut contrôler s'il y a des enregistrements y faisant référence) ou si la table référencée sert à faire des filtrages sélectifs. Dans les autres cas, ce n'est pas forcément très utile.

Fichier

```
select *
from emp e
join dept d on e.deptno=d.deptno
where d.loc='DALLAS'
```

Dans cet exemple, un index sur **d.deptno** est probablement pertinent.

Un autre point important est que généralement, pour accéder à une table, un seul index est utilisé. Donc, si vous avez un prédicat sur la ville et un autre sur l'année, et que ces deux colonnes sont indexées, alors il est probable qu'un seul index soit utilisé (c'est différent s'il y a un **or** entre les deux prédicats).

Ces faits impliquent que contrairement à ce que certains pensent, indexer toutes les colonnes avec un index par colonne n'a généralement pas d'intérêt.

Indexer des colonnes qui ne servent jamais dans des clauses **where** ou **order by** n'a aucun intérêt. Ces index prendront de la place et ne seront jamais utilisés.

Pour le cas des prédicats sur plusieurs colonnes, il faut utiliser des index multicolonnes qui se créent, eux aussi, simplement avec l'instruction suivante :

Fichier

```
create index IS_HabitantsAnVille on habitants (annee, ville);
```

Cet index sera utilisé pour les conditions portant sur les deux colonnes ou sur **annee** seule, mais pas sur **ville** seule. On mettra donc les colonnes dans l'ordre de probabilité d'apparition dans des requêtes avec une partie seulement des colonnes de façon à ce que l'index soit utilisé le plus souvent possible.

Il est par ailleurs possible d'indexer plusieurs fois une même colonne dans des index différents, on pourrait par exemple avoir un autre index sur **ville** et **nom**.

Les index multicolonnes peuvent être utilisés dans un autre cas intéressant, pour faire des index couvrants. Cette technique consiste à mettre toutes les colonnes requises pour une requête dans un index, ainsi, il n'y aura pas à aller chercher la ligne entière dans le tas (ou l'IOT) pour exécuter la requête. L'index suffira ce qui est très performant (mais peut consommer de l'espace si on doit créer plein d'index).

Prenons cet index :

Fichier

```
create index is_deptsal on emp(deptno, sal);
```

Il sera parfaitement adapté à la requête suivante qui utilisera uniquement l'index pour s'exécuter :

Fichier

```
select deptno,avg(sal) from emp group by deptno
```

On mettra en premier dans l'index couvrant : les colonnes utiles pour les filtrages, puis les tris et en dernier les autres colonnes de données utilisées dans la requête.

5. PLAN D'EXÉCUTION

Vous avez mis en place des index et vous vous demandez s'ils sont utilisés dans vos requêtes ? Ou tout simplement, vous vous demandez comment le SGBDR s'y prend pour traiter votre requête ?

SÉLECTIVITÉ ET STATISTIQUES

Dans le cadre de l'évaluation des coûts des requêtes, il faut que l'optimiseur évalue le nombre de lignes manipulées afin d'évaluer quel est le meilleur chemin d'exécution.

Vous imaginez bien que sur une table d'employés, la clause **`NoSecu='17412 3457890'`** ne ramènera pas le même nombre de lignes que **`Region='IDF'`**, nous le savons, mais comment le SGBDR peut-il le savoir ?

Le SGBDR collecte des statistiques sur les tables. Parmi celles-ci, on trouve le nombre d'enregistrements et le nombre de valeurs distinctes pour chaque clé d'index, voire dans certains SGBDR, pour chaque colonne. Le nombre de valeurs nulles, la taille moyenne des données, le minimum et le maximum.

Certains SGBDR (dont MariaDB 10.0.2) ont même des histogrammes qui permettent de pondérer le fait que certaines valeurs sont beaucoup plus présentes que d'autres.

Les commandes suivantes permettent de calculer des statistiques sur les colonnes puis de les visualiser sous MariaDB :

Fichier

```
ANALYZE TABLE emp PERSISTENT FOR ALL;
select * from mysql.column_stats where table_name='emp'
```

Pour voir des statistiques sur la table :

Fichier

```
select * from information_schema.STATISTICS where table_name='emp'
```

Interroger ces tables peut aussi vous apporter des informations intéressantes sur la nature de vos données.

La cardinalité est le nombre d'enregistrements ou de clés distinctes.

La densité est 1/Nombre de valeurs distinctes, (chiffre <=1) c'est la fraction des lignes ramenées par un test d'égalité sur une valeur.

La fréquence est le nombre de valeurs non nulles/nombre de valeurs distinctes, ce chiffre (>=1) donne le nombre moyen d'occurrences pour une valeur donnée.

Ces données vont être utiles pour calculer la sélectivité des prédicats et donc déterminer le nombre de lignes manipulées par une requête.

Par exemple, un prédicat d'égalité tel que **`NoSecu='174123457890'`** sur une liste de personnes aura une forte sélectivité puisque la fréquence est de 1. Du coup, nous aurons une seule ligne sélectionnée par ce prédicat dans une clause **`where`**.

Mais parfois, la fréquence est trompeuse. Par exemple, si vous prenez les habitants de la France, 66 millions de personnes réparties dans 36 000 communes allant de Paris avec 2 229 621 habitants à Rochefourchat qui a un habitant.

… SÉLECTIVITÉ ET STATISTIQUES

La fréquence (moyenne) d'une commune dans notre table est donc de 60000000/36000 soit 1833, donc le prédicat **`Commune='MaCommune'`** devrait en moyenne retourner 1833 lignes, ou vu par la densité devrait retourner 0,0027 % des enregistrements.

Cependant suivant que **`'MaCommune'`** est **`'Rochefourchat'`** ou **`'Paris'`**, on aura en fait une ligne ou plus de 2 millions de lignes.

Sur des valeurs numériques ou temporelles en utilisant les valeurs minimales et maximales, l'optimiseur pourra estimer le nombre de lignes retournées par un prédicat tel que **`AnneeNaissance Between 1916 and 1925`**.

La pyramide des âges français est globalement plate de 2015 à 1945 avec un effectif de 700 000 personnes/an alors qu'en 1920 on est plutôt autour 60 000 personnes. Notre doyenne ayant 113 ans, la fréquence est de 66 millions/113 ans = 584 000. Notre prédicat (sur 10 ans) sera donc estimé à 5,84 millions de lignes, car la fréquence est linéarisée sur 113 ans, alors qu'en fait le résultat sera de 735 000 lignes (soit 8 fois moins que la prédiction), cependant sur les années postérieures à 1945 l'estimation de 584 000 est plutôt juste (la vérité est autour de 700 000).

Le mécanisme d'histogramme permet d'avoir des estimations plus justes sur des valeurs très fréquentes et des répartitions non linéaires de données numériques/temporaires et permet ainsi d'adapter le plan d'exécution en fonction de la valeur passée dans le prédicat.

Pour mettre en œuvre les histogrammes sous MariaDB, exécutez les commandes suivantes avant de lancer **`ANALYZE TABLE`** :

Fichier

```
set histogram_type='double_prec_hb';
set histogram_size=100;
```

Vous trouverez la présentation « ENGINE-INDEPENDENT PERSISTENT STATISTICS WITH HISTOGRAMS IN MARIADB » sur le site de Percona qui explique tout cela en détail.

L'utilisation de valeurs spéciales autres que **`NULL`** peut fausser les estimations, Par exemple, si dans **`AnneeNaissance`** vous mettez des valeurs de **`1900`** à **`2016`** l'optimiseur considèrera que le prédicat **`>2000`** retourne 13 % de la table. Cependant, si vous utilisez la valeur **`0`** (ou **`9999`**) dans certains cas spéciaux (par exemple pour dire que l'information est inconnue), vous allez fausser les estimations sur les intervalles et l'optimiseur considèrera que le prédicat **`>2000`** ramène moins d'un pourcent de la table puisqu'il va considérer que les données sont linéairement reparties entre 0 et 2016, évitez donc cette pratique.

L'instruction **EXPLAIN** (disponible sur la plupart des SGBDR) va vous permettre d'obtenir le plan d'exécution (avec un niveau d'information variable selon les SGBDR).

Fichier

```
mysql> explain extended SELECT empno,ename FROM emp e
    -> WHERE not exists (SELECT 1 FROM emp s WHERE s.mgr=e.empno);
```

Résultat :

Terminal

```
+------------------+---+---+------------+---------+-------+----+--------+-----------+
|  select_type     |tbl|typ|possiblekeys| key     |  ref  |rows|filtered|Extra      |
+------------------+---+---+------------+---------+-------+----+--------+-----------+
|PRIMARY           | e |ALL| NULL       |NULL     |NULL   | 14 |  100.00|Using where|
|DEPENDENT SUBQUERY| s |ref| is_empmgr  |is_empmgr|e.EMPNO|  2 |  100.00|Using index|
+------------------+---+---+------------+---------+-------+----+--------+-----------+
```

Notez que la quantité d'informations disponible sous MySQL est moindre que sur d'autres SGBDR.

À partir de la version 5.6 de MySQL vous pouvez aussi utiliser la trace de l'optimiseur ainsi :

Fichier

```
SET optimizer_trace=" enabled=on";
explain SELECT empno,ename FROM emp e
 WHERE not exists (SELECT 1 FROM emp s WHERE s.mgr=e.empno);
SET optimizer_trace="enabled=off";
SELECT * FROM INFORMATION_SCHEMA.OPTIMIZER_TRACE;
```

Ceci vous donne des détails sur les opérations internes de l'optimiseur (en plus de donner le plan dans cet exemple) au format JSON. Cette fonction me paraît prometteuse, mais donne parfois des résultats qui ne sont pas cohérents avec le plan affiché tout en venant un peu compenser la pauvreté de l'**EXPLAIN**.

CONCLUSION

J'espère vous avoir permis d'éclaircir les choses sur les index, l'essentiel n'est pas compliqué comme vous avez pu le voir... Donc lancez-vous, votre serveur vous en sera reconnaissant (ainsi que vos utilisateurs et l'équipe infrastructure) !

RÉFÉRENCE

[1] Docs MySQL : **http://dev.mysql.com/doc/refman/5.7/en/**

POUR ALLER PLUS LOIN

Si le sujet vous intéresse, je vous invite à regarder mon livre *« Optimisation des bases de données - mise en œuvre sous Oracle»* chez Pearson et qui, comme son nom ne l'indique pas, traite aussi de SQL Serveur et de MySQL. ▉

ÉTAPE 4

4 JE SÉCURISE MA BASE ET L'ACCÈS À MES DONNÉES

À découvrir dans cette partie...

4.1 Sécurisez votre base MariaDB à l'aide de plugins

Le système de plugins de MariaDB permet d'activer de nouvelles fonctionnalités dont certaines dédiées à la sécurisation des données. S'assurer que les mots de passe utilisateurs sont suffisamment robustes, surveiller les accès et les opérations réalisées sur la base, et chiffrer les tables contenant des données sensibles sont les premières précautions à prendre et cela est justement possible grâce à des plugins. p. 80

ÉTAPE 4

JE SÉCURISE MA BASE ET L'ACCÈS À MES DONNÉES

SÉCURISEZ VOTRE BASE MARIADB À L'AIDE DE PLUGINS

Tristan Colombo

Les attaques contre une base de données ne se cantonnent malheureusement pas aux seules injections SQL. MariaDB offre la possibilité d'activer certains *plugins* pour augmenter la sécurité de la base et des données.

MariaDB dispose de nombreux mécanismes permettant de sécuriser les données ou la base elle-même. Ces mécanismes sont directement intégrés, mais pour certains d'entre eux il est nécessaire de les activer à l'aide de *plugins*.

Pour commencer, nous nous pencherons donc sur l'installation de *plugins* MariaDB puis nous testerons trois d'entre eux :

- **Cracklib_Password_Check** qui permet de vérifier la robustesse des mots de passe que vous créez pour vos utilisateurs ;
- **Server_Audit** qui autorise la surveillance des accès aux données de manière avancée ;
- et, enfin, **File_Key_Management** qui donne la possibilité de chiffrer automatiquement des tables.

1. LES PLUGINS

Les *plugins* de MariaDB sont organisés en bibliothèques appelées *plugin_library* qui contiennent le code du *plugin*. Une *plugin_library* est un fichier de bibliothèque dynamique *plugin.so* (so pour *Shared Object*, et sous **Windows** il s'agira bien entendu d'une dll).

Pour installer (ou plutôt activer) un *plugin*, trois méthodes sont disponibles :

- se connecter à la base et utiliser une commande `install library` [1] ou `install soname` [2] ;
- utiliser la ligne de commandes lorsque le serveur de base de données est arrêté : `mysql_plugin <nom_plugin>` [3] suivi de `ENABLE` ou `DISABLE` suivant que l'on souhaite activer ou désactiver le *plugin*.
- ajouter l'argument `--mysql-load-add <nom_plugin>` au lancement du serveur de base de données.

Commençons par nous connecter à la base pour voir quels sont les *plugins* activés :

Terminal

```
$ mysql -p -u root
Enter password:
Welcome to the MariaDB monitor.  Commands end with ; or \g.
...
MariaDB [(none)]> show plugins;
+------------------------------+----------+--------------------+---------+---------+
| Name                         | Status   | Type               | Library | License |
+------------------------------+----------+--------------------+---------+---------+
| binlog                       | ACTIVE   | STORAGE ENGINE     | NULL    | GPL     |
| mysql_native_password        | ACTIVE   | AUTHENTICATION     | NULL    | GPL     |
| mysql_old_password           | ACTIVE   | AUTHENTICATION     | NULL    | GPL     |
| MRG_MyISAM                   | ACTIVE   | STORAGE ENGINE     | NULL    | GPL     |
| CSV                          | ACTIVE   | STORAGE ENGINE     | NULL    | GPL     |
| MEMORY                       | ACTIVE   | STORAGE ENGINE     | NULL    | GPL     |
| MyISAM                       | ACTIVE   | STORAGE ENGINE     | NULL    | GPL     |
| InnoDB                       | ACTIVE   | STORAGE ENGINE     | NULL    | GPL     |
| XTRADB_READ_VIEW             | ACTIVE   | INFORMATION SCHEMA | NULL    | GPL     |
| XTRADB_INTERNAL_HASH_TABLES  | ACTIVE   | INFORMATION SCHEMA | NULL    | GPL     |
| XTRADB_RSEG                  | ACTIVE   | INFORMATION SCHEMA | NULL    | GPL     |
| INNODB_TRX                   | ACTIVE   | INFORMATION SCHEMA | NULL    | GPL     |
...
| INNODB_TABLESPACES_ENCRYPTION | ACTIVE   | INFORMATION SCHEMA | NULL    | BSD     |
| INNODB_TABLESPACES_SCRUBBING  | ACTIVE   | INFORMATION SCHEMA | NULL    | BSD     |
...
| ARCHIVE                      | ACTIVE   | STORAGE ENGINE     | NULL    | GPL     |
```

```
| BLACKHOLE                     | ACTIVE   | STORAGE ENGINE     | NULL    | GPL     |
| Aria                          | ACTIVE   | STORAGE ENGINE     | NULL    | GPL     |
| FEEDBACK                      | DISABLED | INFORMATION SCHEMA | NULL    | GPL     |
| partition                     | ACTIVE   | STORAGE ENGINE     | NULL    | GPL     |
...
+-------------------------------+----------+--------------------+---------+---------+
58 rows in set (0.02 sec)
```

Vous constaterez que bon nombre de *plugins* sont déjà actifs (dans l'exemple ci-dessus, seul **Feedback** est inactif). En plus du statut, le tableau indique trois autres informations :

- le type qui peut être `storage engine`, `authentication`, ou `information schema` ;
- le nom de la bibliothèque (la valeur `NULL` indique un *plugin* intégré ne pouvant pas être désinstallé). Si la valeur est différente de `NULL`, la désinstallation pourra se faire à l'aide des commandes `uninstall library` [4] ou `uninstall soname` [5] ;
- le type de licence (ici uniquement `GPL` et `BSD`).

Il est possible d'obtenir plus d'informations sur un *plugin* spécifique :

Terminal

```
MariaDB [(none)]> select * from information_schema.plugins where plugin_
name='innodb' \G
*************************** 1. row ***************************
           PLUGIN_NAME: InnoDB
        PLUGIN_VERSION: 5.6
         PLUGIN_STATUS: ACTIVE
           PLUGIN_TYPE: STORAGE ENGINE
   PLUGIN_TYPE_VERSION: 100027.0
        PLUGIN_LIBRARY: NULL
PLUGIN_LIBRARY_VERSION: NULL
         PLUGIN_AUTHOR: Oracle Corporation
    PLUGIN_DESCRIPTION: Percona-XtraDB, Supports transactions, row-level
locking, and foreign keys
        PLUGIN_LICENSE: GPL
           LOAD_OPTION: ON
       PLUGIN_MATURITY: Stable
   PLUGIN_AUTH_VERSION: 5.6.31-77.0
1 row in set (0.01 sec)
```

NOTE

L'utilisation de `\G` en fin de requête permet d'obtenir un affichage sous forme de colonnes, beaucoup plus lisible que le tableau obtenu de manière traditionnelle :

Terminal

```
MariaDB [(none)]> select * from information_schema.plugins where plugin_name='innodb';
+------------+---------------+--------------+---------------+--------------------
...
| PLUGIN_NAME | PLUGIN_VERSION | PLUGIN_STATUS | PLUGIN_TYPE | PLUGIN_TYPE_VERSION |
PLUGIN_LIBRARY | PLUGIN_LIBRARY_VERSION | PLUGIN_AUTHOR      | PLUGIN_DESCRIPTION  ...
+-------------+-----------------+---------------------+
| InnoDB      | 5.6             | ACTIVE        | STORAGE ENGINE | 100027.0
| NULL
...
+-------------+-----------------+---------------------+
1 row in set (0.01 sec)
```

2. VÉRIFICATION DE LA ROBUSTESSE DES MOTS DE PASSE

Il faut tout d'abord que vous sachiez que le *plugin* Cracklib_Password_Check n'est apparu qu'à partir de la version 10.1.2 de MariaDB : si vous disposez d'une distribution basée sur Debian et que vous avez utilisé le système de paquets pour installer votre base, il y a de fortes chances pour que votre version de MariaDB soit trop ancienne. Vous avez alors deux solutions :

- installer manuellement MariaDB (reportez-vous à l'article traitant de l'installation de MariaDB dans le présent hors-série) ;
- ajouter une source à votre gestionnaire de paquets pour simplement faire une mise à jour. Cette méthode vous permettra d'obtenir la dernière version stable. À l'heure où ces lignes sont écrites il s'agit de la version 10.1.19, ce qui n'est pas suffisant pour pouvoir tester Cracklib_Password_Check... j'utiliserai donc le dépôt pour la version bêta 10.2 (au moment où vous lirez cet article la version 10.2 sera normalement passée en version stable, sinon, remplacez `10.2` par `10.1` dans la commande `add-apt-repository` ci-dessous si vous souhaitez suivre MariaDB grâce au gestionnaire de paquets sans basculer en 10.2... mais vous ne pourrez pas tester le *plugin*) :

Terminal
```
$ sudo apt-get install -y software-properties-common
$ sudo apt-key adv --recv-keys --keyserver keyserver.ubuntu.com 0xcbcb082a1bb943db
$ sudo add-apt-repository 'deb [arch=amd64,i386] http://nyc2.mirrors.digitalocean.
com/mariadb/repo/10.2/debian jessie main'
$ sudo apt-get update
$ sudo apt install mariadb-server
```

Cette dernière commande supprimera votre ancienne version de MariaDB pour installer la nouvelle (vous ne perdrez pas vos tables... si tout se passe bien). Vous pourrez tester que vous disposez bien désormais de la version 10.2 par :

Terminal
```
$ mysql --version
mysql  Ver 15.1 Distrib 10.2.2-MariaDB, for debian-linux-gnu (x86_64) using readline 5.2
```

Comme nous l'avons vu, en théorie l'installation du *plugin* Cracklib_Password_Check est très simple... à condition que la bibliothèque cracklib soit installée !

Terminal
```
$ sudo apt install cracklib-runtime libcrack2
```

Pour vous assurer que tout fonctionne correctement, vérifiez que votre variable d'environnement `PATH` a bien accès au répertoire `/usr/sbin` puis créez un dictionnaire de mots de passe pour cracklib :

Terminal
```
$ sudo create-cracklib-dict /usr/local/share/cracklib/cracklib-small
51526 51526
```

Vous pouvez maintenant tester la robustesse des mots de passe en ligne de commandes :

Terminal
```
$ echo "password" | cracklib-check
password: basé sur un mot du dictionnaire
$ echo "1234567" | cracklib-check
1234567: trop simple/systématique
$ echo "bof" | cracklib-check
bof: BEAUCOUP trop court
```

ÉTAPE 4

Passons maintenant à l'installation dans MariaDB pour utiliser ce mécanisme :

Terminal

```
MariaDB [(none)]> install soname 'cracklib_password_check';
Query OK, 0 rows affected (0.05 sec)

MariaDB [(none)]> set password = password('azerty');
ERROR 1819 (HY000): Your password does not satisfy the current policy requirements
```

NOTE

Si la version de votre base est inférieure à 10.2, mais supérieure à 10.1.2, et que vous ne souhaitez pas la mettre à jour, vous pouvez utiliser le *plugin* Simple_Password_Check :

Terminal

```
MariaDB [(none)]> install soname 'simple_password_check';
Query OK, 0 rows affected (0.04 sec)

MariaDB [(none)]> set password = password('123');
ERROR 1819 (HY000): Your password does not satisfy the current policy requirements
```

Ce *plugin* se contente de s'assurer que le mot de passe contient un nombre spécifique de caractères, chiffres, etc. Par défaut, il s'agit d'un mot de passe d'un minimum de huit caractères avec au moins un chiffre, une lettre majuscule, une lettre minuscule et un caractère qui ne soit ni un chiffre, ni une lettre.

Il est possible de configurer le comportement de ce *plugin* à l'aide des variables suivantes (qui acceptent des valeurs allant de **0** à **1000**) :

- ⇨ **simple_password_check_minimal_length** : taille minimale du mot de passe ;
- ⇨ **simple_password_check_letters_same_case** : nombre minimal de lettres en majuscules et en minuscules (donc autant de lettres majuscules que minuscules, c'est un peu dommage...) ;
- ⇨ **simple_password_check_digits** : nombre de chiffres ;
- ⇨ **simple_password_check_other_characters** : nombre de caractères autres que des chiffres ou des lettres.

Ces variables étant globales, leur modification se fera par une commande du type :

Terminal

```
MariaDB [(none)]> set global simple_password_check_digits=2;
```

Vous pourrez voir les valeurs de ces variables par :

Terminal

```
MariaDB [(none)]> select * from information_schema.system_variables where
variable_name like 'simple_password %' \G
*************************** 1. row ***************************
       VARIABLE_NAME: SIMPLE_PASSWORD_CHECK_DIGITS
      SESSION_VALUE: NULL
       GLOBAL_VALUE: 2
GLOBAL_VALUE_ORIGIN: SQL
      DEFAULT_VALUE: 1
     VARIABLE_SCOPE: GLOBAL
      VARIABLE_TYPE: INT UNSIGNED
   VARIABLE_COMMENT: Minimal required number of digits
   NUMERIC_MIN_VALUE: 0
   NUMERIC_MAX_VALUE: 1000
  NUMERIC_BLOCK_SIZE: 1
     ENUM_VALUE_LIST: NULL
        READ_ONLY: NO
COMMAND_LINE_ARGUMENT: REQUIRED
*************************** 2. row ***************************
       VARIABLE_NAME: SIMPLE_PASSWORD_CHECK_LETTERS_SAME_CASE
...
```

3. SURVEILLANCE DES ACCÈS AUX DONNÉES

Un autre *plugin* permet non plus de vraiment sécuriser la base, mais d'être en mesure de savoir ce qui s'y passe en surveillant qui s'y connecte, quelles sont les requêtes exécutées, les tables modifiées, etc. Ceci est déjà possible grâce au système de log de la base, mais ce *plugin* permet une configuration beaucoup plus poussée.

Pour commencer, comme tout *plugin* il faut l'activer :

Terminal
```
MariaDB [(none)]> install plugin server_audit soname 'server_audit.so';
Query OK, 0 rows affected (0.02 sec)
```

Ensuite, il faut modifier les valeurs de certaines variables pour activer les logs et indiquer quels événements surveiller. Pour cela, on utilise `server_audit_logging` et `server_audit_events`, mais d'autres variables sont disponibles [6], notamment pour activer la rotation des fichiers de log comme nous allons le voir. Nous allons modifier ces variables depuis la base, mais il est possible de les définir dans le fichier `my.cnf` dans la section `[mysqld]` :

Terminal
```
MariaDB [(none)]> set global server_audit_logging=on;
Query OK, 0 rows affected (0.00 sec)
MariaDB [(none)]> set global server_audit_events='connect,query,table';
Query OK, 0 rows affected (0.00 sec)
```

Par défaut, les logs seront inscrits dans `/var/log/syslog`. Nous allons définir un fichier spécifique (`/var/log/mysql/audit.log`) et activer la rotation des fichiers de log :

Terminal
```
MariaDB [(none)]> set global server_audit_output_type='file';
Query OK, 0 rows affected (0.00 sec)
MariaDB [(none)]> set global server_audit_file_path='/var/log/mysql/
audit.log' ;
Query OK, 0 rows affected (0.00 sec)
MariaDB [(none)]> set global server_audit_logging=on;
Query OK, 0 rows affected (0.00 sec)
MariaDB [(none)]> set global server_audit_file_rotate_size=1000000;
Query OK, 0 rows affected (0.00 sec)
MariaDB [(none)]> set global server_audit_file_rotations=6;
Query OK, 0 rows affected (0.00 sec)
```

Nous avons défini ici `6` fichiers de rotation d'une taille de `1000000` d'octets. Après quelques opérations, vous verrez le fichier `/var/log/mysql/audit.log` se remplir :

Terminal
```
$ tail /var/log/mysql/audit.log
20161130 15:40:13,ma-machine,root,localhost,6,174,QUERY,mysql,'select * from
user where User like \'%root%\'',0
20161130 15:47:00,ma-machine,root,localhost,6,175,QUERY,mysql,'set global
server_audit_output_type=\'file\'',0
20161130 15:51:39,ma-machine,root,localhost,6,176,QUERY,mysql,'set global
server_audit_file_rotate_size=1000000',0
20161130 15:51:54,ma-machine,root,localhost,6,177,QUERY,mysql,'set global
server_audit_file_rotations=6',0
```

À retenir

Sachez que si vous installez plusieurs *plugins* de test de robustesse des mots de passe, vos mots de passe devront être acceptés par tous les *plugins* pour être valides.

```
20161130 15:53:30,ma-machine,root,localhost,6,0,DISCONNECT,mysql,,0
20161130 15:53:34,ma-machine,root,localhost,7,0,CONNECT,,,0
20161130 15:53:34,ma-machine,root,localhost,7,179,QUERY,,'select @@version_
comment limit 1',0
20161130 15:53:41,ma-machine,root,localhost,7,180,QUERY,,'SELECT DATABASE()',0
20161130 15:53:41,ma-machine,root,localhost,7,182,QUERY,test_audit,'show
databases',0
20161130 15:53:41,ma-machine,root,localhost,7,183,QUERY,test_audit,'show
tables',0
```

À savoir

Il est possible d'exclure des utilisateurs de la surveillance à l'aide de `server_audit_excl_users`.

4. VOUS STOCKEZ DES DONNÉES SENSIBLES… AVEZ-VOUS PENSÉ À LES CHIFFRER ?

Si vous stockez des données sensibles, il est essentiel de les chiffrer. MariaDB ne pourra rien faire pour ce qui est du transfert des données et il faudra bien entendu penser à passer par un protocole sécurisé (ssl) ; par contre, pour ce qui est des tables, vous pouvez activer le chiffrement à la création de la table ou a posteriori avec `alter`.

Il faudra installer ici le *plugin* File_Key_Management :

Terminal

```
MariaDB [(none)]> install soname 'file_key_management';
Query OK, 0 rows affected (0.00 sec)
```

Encore une fois, ceci peut être fait depuis le fichier `my.cnf` et l'on peut également définir le fichier contenant les informations de cryptage :

Fichier

```
...
plugin-load-add = file_key_management
file_key_management_filename = /var/lib/mysql/keys.txt
...
```

Pour créer le fichier `/var/lib/mysql/keys.txt`, utilisez les informations `iv` (*initialization vector*) et `key` fournies par la commande `openssl` :

Terminal

```
$ openssl enc -aes-256-cbc -P -md sha1
enter aes-256-cbc encryption password:
Verifying - enter aes-256-cbc encryption password:
salt=DD9C17EB8DB13B1B
key=7DE68095D70F2CA3B81B0497A30E1E57E1351B238FE66A12D7398B6D227E014C
iv =F49C44D37A20C220B241659535AC9528
```

Le format des lignes de `keys.txt` est `identifiant;initialization vector;key`. D'après les données obtenues, nous pouvons créer le fichier :

Fichier

```
1;F49C44D37A20C220B241659535AC9528;7DE68095D70F2CA3B81B0497A30E1E5
7E1351B238FE66A12D7398B6D227E014C
```

Le chiffrement est alors très simple puisqu'il suffit de spécifier **encrypted=yes** :

Terminal

```
MariaDB [(none)]> create table tab_1 (id int not null primary key, secret
varchar(150)) encrypted=yes;
Query OK, 0 rows affected (0.22 sec)
...
MariaDB [(none)]> alter table tab_2 encrypted=yes;
Query OK, 3 rows affected (0.36 sec)
Records: 3  Duplicates: 0  Warnings: 0
```

NOTE

Si vous voulez utiliser plusieurs clés de chiffrement (définies dans le même fichier **keys.txt** avec des identifiants différents), vous pouvez le spécifier à l'aide de **encryption_key_id**.

Prenons l'exemple du fichier **keys.txt** suivant :

Fichier

```
1;F49C44D37A20C220B241659535AC9528;238FE66A12D7398B6D...
2;AA98CFE37A20C220FFD43E6785AC9528;F2CA3B81B0497A30E1...
3;45BAD65CA20D91023F8930D35AC952DE;0F2CA3B81B0497A30E...
...
```

Alors il est possible de choisir la clé de chiffrement par :

Terminal

```
MariaDB [(none)]> create table tab_1 (id int not null primary key, secret
varchar(150)) encrypted=yes encryption_key_id=2;
```

CONCLUSION

Dans cet article, nous avons pu avoir un aperçu de certains *plugins* de sécurisation de MariaDB. Il est évident que la mise en place de ceux-ci ne garantira pas que votre base soit préservée de toute attaque, mais ils constituent en quelque sorte les précautions minimales à prendre, conjuguées bien entendu à une attention particulière aux vulnérabilités aux injections SQL de vos programmes... ▮

RÉFÉRENCES

[1] Commande **install library** : **https://mariadb.com/kb/en/mariadb/install-plugin/**

[2] Commande **install soname** : **https://mariadb.com/kb/en/mariadb/install-soname/**

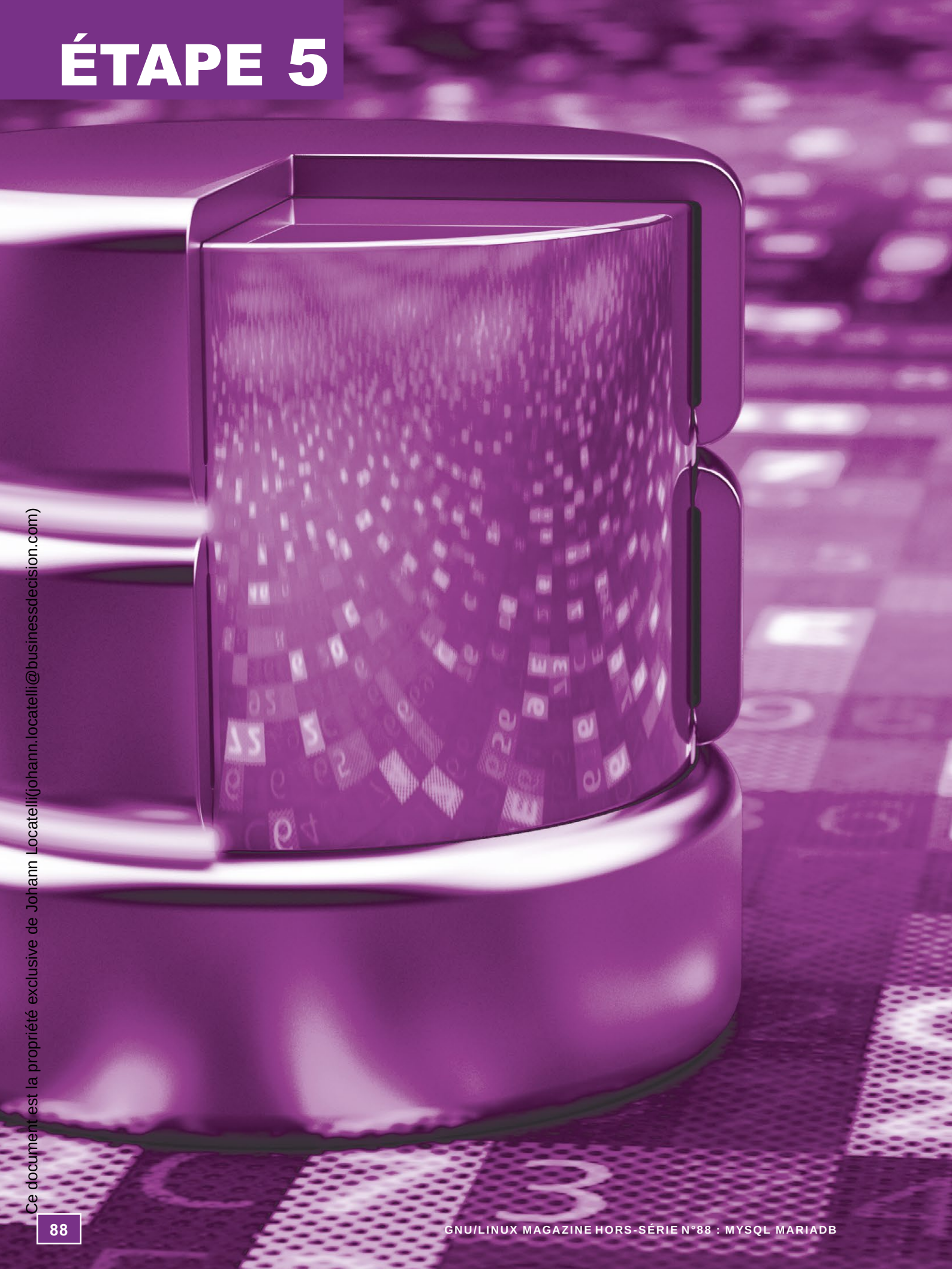
[3] Commande **mysql_plugin** : **https://mariadb.com/kb/en/mariadb/mysql_plugin/**

[4] Commande **uninstall plugin** : **https://mariadb.com/kb/en/uninstall-plugin/**

[5] Commande **uninstall soname** : **https://mariadb.com/kb/en/mariadb/uninstall-soname/**

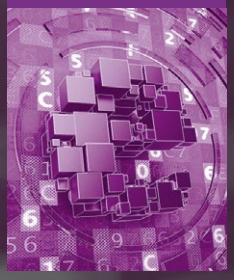
[6] Variables de configuration du *plugin* Server_Audit : **https://mariadb.com/kb/en/mariadb/server_audit-system-variables/**

ÉTAPE 5



5 JE CRÉE DES PROGRAMMES UTILISANT MES BASES

À découvrir dans cette partie...



5.1 Accédez à vos données en C++ avec sqlpp11

Les traitements complexes sur les informations contenues dans les bases de données relationnelles nécessitent l'utilisation d'un langage de programmation plus général que ne l'est SQL. Nous nous intéresserons ici au langage C++. p. 90

5.2 Utilisez MySQL avec l'API REST et Python

Afin d'utiliser MySQL pour en faire un backend dit RESTFul, et donc compatible avec beaucoup de nouveaux frameworks web orientés REST, nous devons jouer un peu avec MySQL. p. 104

5.3 Découvrez les nouvelles fonctions natives SQL pour manipuler du contenu JSON

Depuis la version 5.7.8 de MySQL, il est possible de gérer des documents JSON en base. p. 114

ÉTAPE 5

JE CRÉE DES PROGRAMMES UTILISANT MES BASES

ACCÉDEZ À VOS DONNÉES EN C++ AVEC SQLPP11

Yves Bailly

Si les outils présentés précédemment permettent déjà de faire pas mal de choses avec votre base de données, celle-ci ne montrera véritablement toute sa puissance qu'au travers de programmes effectuant des traitements complexes, des calculs sophistiqués, sur les données extraites.

Lorsqu'on doit interagir avec une base de données relationnelle dans un langage typé comme le C ou le C++, il n'est pas rare que le *framework* utilisé s'appuie sur un typage « faible » pour récupérer les données, à base de types variables comme `QVariant`, ou pire de pointeurs génériques comme `void*`. Et nombre de ces *frameworks* imposent au programmeur de construire « à la main » ses requêtes en composant les commandes SQL dans une chaîne de caractères.

Une approche parfaitement contraire a été choisie pour la bibliothèque **sqlpp11 [1]**, où le « 11 » est le signal de l'utilisation de ce que l'on nomme le « C++ moderne », né de la révolution qu'a été le standard C++ édité en 2011. Cette petite bibliothèque permet de construire des requêtes en utilisant une syntaxe C++, le SQL « pur » étant complètement encapsulé. De plus, elle permet d'utiliser de « vrais » types, comme `int` ou `std::string`, au travers d'un véritable méta-modèle en C++ du schéma de la base : les incohérences entre le code et la base de données sont alors détectées dès la compilation, évitant de longues heures de test et de débogage.

1. PRÉPARATION

Sqlpp11 n'a que peu de dépendances : **Boost [2]** pour la partie C++ (notamment les programmes d'exemple), et le module **Python Pyparsing [3]** qui offre une alternative au fameux couple **lex** et **yacc** pour la génération de grammaires. Naturellement, les fichiers de développement de **MariaDB** sont également nécessaires, ainsi que **CMake [4]** pour la compilation. Des paquetages pour satisfaire ces dépendances sont disponibles pour la plupart des distributions majeures. Par exemple, pour une distribution basée sur **Ubuntu** :

Terminal
```
$ sudo apt-get install libboost-all-dev python-pyparsing libmariadb-client-
lgpl-dev cmake
```

Par ailleurs, il vous faudra bien évidemment un compilateur C++ qui « comprenne » le standard C++11. Un **GCC** ou un **Clang** de moins de cinq ans devrait faire l'affaire.

1.1 Codes sources

Le reste des codes source est hébergé sur **GitHub**. La méthode naturelle pour les récupérer est donc d'utiliser `git`. On commence par une bibliothèque C++ de manipulation de dates :

Terminal
```
$ git clone https://github.com/HowardHinnant/date.git
```

Puis le code de sqlpp11 lui-même, avec le connecteur **MySQL** :

Terminal
```
$ git clone https://github.com/rbock/sqlpp11.git
$ git clone https://github.com/rbock/sqlpp11-connector-mysql.git
```

Si vous n'êtes intéressé que par les codes sources, sans tout l'historique git, vous pouvez simplement obtenir des archives ainsi :

Terminal

```
$ wget -nd https://github.com/HowardHinnant/date/archive/master.zip -O date_
master.zip
$ wget -nd https://github.com/rbock/sqlpp11/archive/master.zip -O sqlpp11_
master.zip
$ wget -nd https://github.com/rbock/sqlpp11-connector-mysql/archive/master.
zip -O sqlpp11-connector-mysql_master.zip
```

Dans la suite, nous supposerons que les trois sources précédents sont placés dans un même répertoire, dans des sous-répertoires nommés **`date`**, **`sqlpp11`** et **`sqlpp11-connector-mysql`**, et que le chemin de ce répertoire est stocké dans une variable d'environnement **`$SQLPP11`**, par exemple :

Terminal

```
$ export SQLPP11=/home/yves/Programs/sqlpp11
```

1.2 Compilation

Comme évoqué plus haut, la compilation des différents composants de sqlpp11 s'effectue simplement avec CMake. Pour compiler sqlpp11, placez-vous dans le dossier correspondant, puis exécutez :

Terminal

```
$ cd $SQLPP11/sqlpp11
$ cmake -DHinnantDate_ROOT_DIR=$SQLPP11/date/ -DHinnantDate_INCLUDE_
DIR=$SQLPP11/date/ .
$ make
```

La compilation du connecteur pour MariaDB/MySQL est tout aussi difficile :

Terminal

```
$ cd $SQLPP11/sqlpp11-connector-mysql
$ cmake -DMYSQL_INCLUDE_DIR=/usr/include/mariadb/ .
$ make
```

Cela va également compiler des programmes de démonstration et de test.

2. LE PROBLÈME ET LA BASE DE TEST

Pour expérimenter avec sqlpp11, il nous faut une base de données, avec des données. Nous allons prendre comme exemple un problème inspiré d'un épisode de la série *Science Étonnante*, diffusée sur **YouTube** par David Louapre **[5]** : il s'agit de créer un algorithme générateur de mots, qui « sonnent vrais » dans une langue donnée. Un assemblage parfaitement aléatoire de lettres n'a aucune chance d'aboutir : par exemple, la séquence aléatoire « **`ajbwxewq`** » ne ressemble à rien en français (ni dans la plupart des langues d'ailleurs). Par contre, la séquence plus contrôlée « **`forabiens`** » est plus crédible.

L'algorithme s'appuie sur une analyse statistique des mots existants d'une langue : pour chaque lettre de l'alphabet, on mesure la fréquence avec laquelle elle suit une autre lettre de l'alphabet. Ainsi, on verra qu'un **`'b'`** suivi d'un **`'w'`** est très rare en français, tandis

qu'un `'r'` suivi d'un `'a'` est bien plus fréquent. À partir de cette information et de quelques autres, il devient possible de générer des mots crédibles. Nous n'irons pas jusqu'à cette génération dans cet article, mais c'est un prétexte intéressant.

Notre base de données va être utilisée pour stocker l'information de la fréquence avec laquelle une lettre en suit une autre, dans une langue donnée. Ce qui nous donne le schéma très simple de la Figure 1.

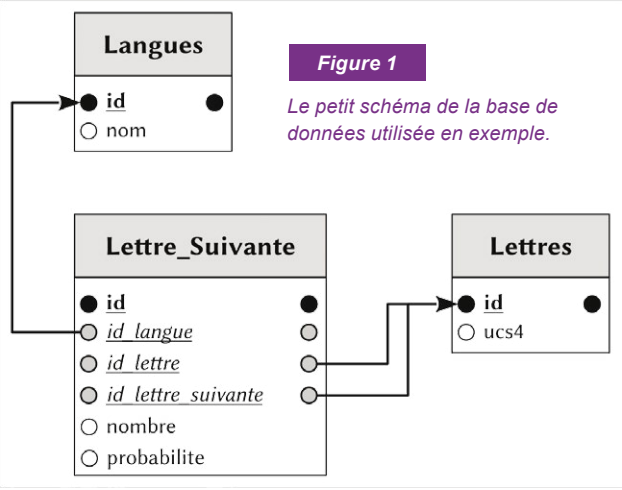


Figure 1

Le petit schéma de la base de données utilisée en exemple.

Les lettres seront stockées en utilisant leur code Unicode, sur 32 bits (codage UCS4). C'est évidemment un peu artificiel : plutôt que d'avoir une table dédiée, nous aurions pu utiliser directement ce code dans la table **`Lettre_Suivante`**. Mais cela permet d'avoir une structure un poil moins triviale pour expérimenter.

Les tables sont créées à l'aide des commandes SQL suivantes :

```
CREATE TABLE Langues (
    id INT UNSIGNED
        NOT NULL
        AUTO_INCREMENT
        PRIMARY KEY,
    nom VARCHAR(255)
);
```

```
CREATE TABLE Lettres (
    id INT UNSIGNED
        NOT NULL
        AUTO_INCREMENT
        PRIMARY KEY,
    ucs4 INT UNSIGNED
);
```

```
CREATE TABLE Lettre_Suivante (
    id INT UNSIGNED
        NOT NULL
        AUTO_INCREMENT
        PRIMARY KEY,
    id_langue INT UNSIGNED NOT NULL,
    id_lettre INT UNSIGNED NOT NULL,
    id_lettre_suivante INT UNSIGNED NOT NULL,
    nombre INT UNSIGNED,
    probabilite DOUBLE,
    CONSTRAINT constr_id_langue
        FOREIGN KEY (id_langue)
        REFERENCES Langues(id)
        ON DELETE CASCADE,
    CONSTRAINT constr_id_lettre
        FOREIGN KEY (id_lettre)
        REFERENCES Lettres(id)
        ON DELETE CASCADE,
    CONSTRAINT constr_id_lettre_suivante
        FOREIGN KEY (id_lettre_suivante)
        REFERENCES Lettres(id)
        ON DELETE CASCADE
);
```

Rien de bien compliqué. Mais pour simpliste qu'il paraisse, ce problème et le modèle de données correspondant soulèvent plusieurs défis :

- Le premier est le remplissage de la base de données : effectuer un décompte « à la main » n'est humainement pas possible, surtout si on considère plusieurs langues. Les insertions dans la base doivent donc être automatisées, programmées.
- Ensuite, la production d'une sorte de rapport visuellement informatif, comme les tableaux proposés dans la vidéo originale (à 3'35), va très certainement au-delà des capacités de tous les générateurs de rapports usuels. Une programmation dédiée est là aussi nécessaire.
- Enfin, la réponse même au problème, à savoir la génération de mots crédibles, paraît difficilement réalisable en n'utilisant que le langage SQL – même si c'est sans doute possible.

À retenir

La simple information de succession ne permet pas d'obtenir des résultats de bonne qualité, comme l'indique l'article qui accompagne la vidéo [5] ainsi que le complément ultérieur [6] : nous n'obtiendrions pas quelque chose de probant. Notre modèle est trop simpliste, mais cela sera suffisant pour nos expérimentations.

La façon la plus efficace de relever ces défis est sans doute d'utiliser un langage de programmation généraliste, comme le C++, en exploitant le contenu de la base de données via une interface comme celle proposée par sqlpp11. Dans la suite, nous supposerons que le modèle précédent est implémenté dans une base de données nommée **lettre_suivante**.

3. PRÉPARATION

La première opération consiste à extraire une description de la base de données, sous la forme d'un script SQL permettant de la recréer – c'est-à-dire une succession de clauses **CREATE TABLE** plus quelques autres informations. Le plus simple est sans doute d'utiliser l'utilitaire **mysqldump**, par exemple :

Terminal
```
$ mysqldump --no-data lettre_suivante > lettre_suivante.sql
```

Ceci nous donne dans le répertoire courant un fichier **lettre_suivante.sql**, qui contient l'information voulue : une description complète de la structure de la base de données. À partir de là, le script Python **ddl2cpp** fournit par sqlpp11 génère un fichier d'en-tête C++ reprenant les descriptions des tables de la base de données, sous la forme de classes et structures. Dans notre exemple, cela donne :

Terminal
```
$ $SQLPP11/sqlpp11/scripts/ddl2cpp lettre_suivante.sql ./ls_db LS
```

Le premier paramètre est le chemin vers le fichier de description tel qu'obtenu il y a un instant par **mysqldump**. Le deuxième est le chemin vers le fichier d'en-tête que l'on souhaite générer – **sans** l'extension **.h**, celle-ci sera ajoutée automatiquement. Ici, nous allons donc obtenir le fichier **ls_db.h** dans le répertoire courant. Enfin, le troisième et dernier paramètre est le nom d'un espace de nommage (**namespace**) C++ dans lequel seront confinées toutes les déclarations de types générées. Ici, ce sera **LS**.

4. CONNEXION

Il est temps d'écrire notre programme, et de commencer par nous assurer qu'il est possible de nous connecter à la base de données. Pour cet exemple simple, nous allons tout faire tenir dans un fichier unique **ls_db.cpp**, situé dans le même répertoire que l'en-tête **ls_db.h** que nous venons de générer.

Pour simplifier la compilation du programme, copiez l'archive obtenue lors de la compilation du connecteur dans le répertoire courant :

Terminal
```
$ cp $SQLPP11/sqlpp11-connector-mysql/src/libsqlpp-mysql.a .
```

La compilation se fait alors avec :

Terminal
```
$ g++ -std=c++11 -Wall -O3 \
   -I$SQLPP11/sqlpp11/include \
   -I$SQLPP11/sqlpp11-connector-mysql/include/ \
```

```
    -I$SQLPP11/date \
    ls_db.cpp \
    -L. -lsqlpp-mysql -lmariadb \
    -o ls_db
```

Remarquez les trois chemins d'inclusion, pour accéder aux déclarations nécessaires. Notamment les en-têtes de sqlpp11 :

Fichier

```
#include <sqlpp11/sqlpp11.h>
#include <sqlpp11/mysql/mysql.h>
#include "ls_db.h"
namespace mdb = sqlpp::mysql;
```

La première ligne inclue l'en-tête générique de sqlpp11, tandis que la suivante inclue l'en-tête spécifique pour le connecteur MySQL. L'en-tête pour notre base de données est utilisé ensuite. Et pour nous simplifier l'écriture, on se donne un *namespace* au nom plus simple pour utiliser les outils du connecteur MySQL.

Commençons par nous connecter à la base de données :

Fichier

```
auto conn_cfg = std::make_shared<mdb::connection_config>();
conn_cfg->host = "127.0.0.1";
conn_cfg->user = "lettre_suivante";
conn_cfg->password = "lettre_suivante";
conn_cfg->database = "lettre_suivante";
conn_cfg->debug = true;
mdb::connection db(conn_cfg);
```

Les premières lignes permettent de configurer la connexion. Les chaînes de caractères de la structure **`sqlpp::mysql::connection_config`** sont toutes de type **`std::string`**. La connexion en elle-même est établie lors de la création de l'instance de **`sqlpp::mysql::connection`**, à la dernière ligne.

Si la connexion échoue (mauvais utilisateur, mauvais mot de passe, etc.) une exception de type **`sqlpp::exception`** est levée. Un « vrai » code en production devrait sans doute encadrer la création de la variable **`db`** dans une clause **`try...catch`**.

Mais si tout se passe bien, nous pouvons dès maintenant utiliser notre base de données en code C++.

5. INSERTIONS

La connaissance, pour une langue donnée, de la fréquence avec laquelle une lettre en suit une autre, nécessite d'analyser un corpus de textes dans cette langue : ce sera notre matière brute. Le programme va alors attendre en premier paramètre une langue, suivant des chemins vers un ou plusieurs fichiers textes (encodés en UTF8). Comme source pour le français, nous allons tout simplement utiliser les fichiers des « *Misérables* » de Victor Hugo, obtenus à partir du projet **Gutenberg [7]**. Chaque ligne est lue, les mots séparés, et chaque mot unique est parcouru lettre à lettre. Les détails de cette opération n'ont ici pas beaucoup d'importance. Disons simplement que nous allons nous appuyer sur **QtCore [8]** pour les facilités de traitement de chaînes et des encodages, et que nous allons remplir le double dictionnaire suivant :

Fichier

```
std::unordered_map<char32_t, std::unordered_map<char32_t, size_t>> successions;
```

Pour chaque lettre (première clef **char32_t**), on stocke la liste des lettres qui la suivent (seconde clef **char32_t**) avec le nombre de fois que cette succession est rencontrée (la donnée **size_t**).

On suppose ici démarrer avec une base vide, et que le double dictionnaire précédent est rempli. Grâce aux déclarations générées plus haut dans **ls_db.h**, nous pouvons obtenir des variables qui représentent chacune des tables de notre base de données :

Fichier

```
auto const Langues = LS::Langues{};
auto const Lettres = LS::Lettres{};
auto const Lettre_Suivante = LS::LettreSuivante{};
```

Remarquez l'utilisation du *namespace* **LS** que nous avions donné à **ddl2cpp**. Remarquez également que le caractère de soulignement **_** a disparu du nom de la table **Lettre_Suivante**... c'est un petit défaut de sqlpp11, qui ne conserve que les caractères alphanumériques dans les noms des éléments (tables ou champs).

Nous pouvons alors insérer la langue, qui nous a été donnée en premier paramètre :

Fichier

```
size_t langue_id = db(sqlpp::insert_into(Langues).set(Langues.nom = argv[1]));
```

Beaucoup de choses se passent dans cette simple ligne.

Pour rappel, **db** est l'instance de **sqlpp::mysql::connection** créée précédemment pour représenter la connexion à la base de données. Cette classe fournit (entre autres) un **operator()**, qui permet d'exécuter des requêtes SQL.

La requête est ici générée par un appel à **sqlpp::insert_into()**, dont le rôle est justement de générer une requête SQL d'insertion de données : en paramètre est attendue la variable représentant la table dans laquelle insérer (ici, **Langues**). Ensuite, chaque champ de la table apparaît comme un champ de la structure qui représente la table : il suffit d'affecter une valeur à chacun (ici, **Langues.nom = argv[1]**), le tout passé en paramètre d'une fonction **set()** présente dans le type retourné par **sqlpp::insert_into()** - dont la découverte de la nature exacte est laissée en exercice au lecteur.

Notre table **Langues** comporte opportunément une colonne **id** à incrément automatique (**AUTO_INCREMENT** dans la déclaration de la table en SQL). Dans ce cas, l'exécution de l'insertion nous retourne l'identifiant nouvellement créé sous la forme d'un **size_t**, ce qui est bien pratique. S'il n'y a pas de colonne de cette nature, alors **0** est retourné. On conserve cet identifiant dans la variable **langue_id**.

Nous allons exploiter cela pour remplir la table des lettres et la table des successions. Pour éviter des requêtes incessantes, on se donne une structure pour mémoriser l'identifiant donné à chaque lettre :

Fichier

```
std::unordered_map<char32_t, size_t> lettres_ids;
```

Puis on parcourt notre double dictionnaire :

Fichier

```
for(auto const& lettre_src: successions)
{
   char32_t const lettre = lettre_src.first;
   if ( lettres_ids.find(lettre) == lettres_ids.end() )
   {
      lettres_ids[lettre] = db(
         sqlpp::insert_into(Lettres)
         .set(Lettres.ucs4 = lettre)
         );
   }
   size_t const id_lettre = lettres_ids[lettre];
```

Pour chaque lettre « source », on vérifie si on connaît déjà son identifiant. Si ce n'est pas le cas, on l'ajoute à la table **Lettres**, avec une syntaxe similaire à celle utilisée plus haut pour la table des langues. Puis on conserve cet identifiant pour la suite :

Fichier

```
    auto const& suivantes = lettre_src.second;
    for(auto const& suivante: suivantes)
    {
        char32_t const lettre_suiv = suivante.first;
        if ( lettres_ids.find(lettre_suiv) == lettres_ids.end() )
        {
            lettres_ids[lettre_suiv] =
                    db(sqlpp::insert_into(Lettres)
                       .set(Lettres.ucs4 = lettre_suiv));
        }
        size_t const id_lettre_suivante = lettres_ids[lettre_suiv];
        db(sqlpp::insert_into(Lettre_Suivante).set(
               Lettre_Suivante.idLangue = id_langue,
               Lettre_Suivante.idLettre = id_lettre,
               Lettre_Suivante.idLettreSuivante = id_lettre_suivante,
               Lettre_Suivante.nombre = suivante.second,
               Lettre_Suivante.probabilite = 0.0
               ));
    }
}
```

La suite consiste à parcourir les lettres « suivantes » de la lettre source. À nouveau, chacune d'elle est insérée dans la table **Lettres** si besoin. Et enfin, on insère cette succession dans la table **Lettre_Suivante**.

La valeur de chaque champ est donnée par une simple affectation, les affectations étant simplement listées, l'une après l'autre, dans l'appel à **set()**. Les *variadic templates* (voir *GNU/Linux Magazine n°159*, *160* et *161*) sont à l'œuvre. Remarquez à nouveau comme les _ ont disparu des noms des champs, au profit d'un nommage en « lower Camel Case » : la première lettre de l'identificateur est en minuscule, les initiales des mots suivants (qui suivaient les _ disparus) en majuscule. Par exemple, le champ SQL **id_lettre_suivante** est devenu **idLettreSuivante**.

Si vous regardez les codes précédents, vous pourrez constater que nous utilisons les types C++ usuels le plus naturellement du monde. C'est là peut-être la force principale de sqlpp11 : l'intégrité des types est conservée, et l'écriture est relativement naturelle pour du C++. Considérons par exemple le champ **probabilite**, qui est déclaré de type **DOUBLE** en SQL. Si par mégarde nous avions tenté d'y inscrire une valeur d'un type incompatible, comme une chaîne de caractères, le compilateur nous aurait gratifiés d'une insulte dans ce genre-là :

Terminal

```
../sqlpp11/include/sqlpp11/column.h: In instantiation of 'sqlpp::assignment_
t<sqlpp::column_t<Table, ColumnSpec>, typename sqlpp::wrap_operand<T,
void>::type> sqlpp::column_t<Table, ColumnSpec>::operator=(T) const [with T =
const char*; Table = LS::LettreSuivante; ColumnSpec = LS::LettreSuivante_::Pro
babilite; typename sqlpp::wrap_operand<T, void>::type = sqlpp::text_operand]':
../lettre_suivante/ls_db.cpp:120:48:   required from here
../sqlpp11/include/sqlpp11/column.h:90:7: error: static assertion failed:
invalid rhs assignment operand
      static_assert(_is_valid_assignment_operand<rhs>::value, "invalid rhs
assignment operand");
      ^
```

Ce qui n'est pas forcément parlant à première vue, mais nous évite d'exécuter un programme incorrect.

De même, si le schéma de la base de données venait à évoluer (ajout ou suppression d'une table, d'une colonne, modification d'un type, etc.), il suffit d'appliquer à nouveau **`mysqldump`** et **`ddl2cpp`**, puis de recompiler le programme. Toute opération invalide (référence à une colonne disparue, mauvais type ...) sera immédiatement détectée dès la compilation, ce qui est bien plus efficace et bien moins coûteux que de ne la découvrir qu'en phase de test, ou pire, par un bug chez l'utilisateur final.

6. SÉLECTION ET CALCULS

Le lecteur attentif aura sans doute remarqué un léger souci dans l'insertion précédente dans la table **`Lettre_Suivante`** : la valeur donnée au champ **`probabilite`** est statique, **`0.0`**. Ce qui n'est pas ce que nous voulons. Pour chaque couple **`(lettre, lettre_suivante)`**, cette valeur devrait être le nombre d'occurrences de la **`lettre_suivante`** pour cette **`lettre`** (le champ **`nombre`**) divisé par la somme des occurrences de toutes les **`lettre_suivante`**s pour cette **`lettre`**.

Nous aurions pu effectuer ce calcul à partir du double dictionnaire, mais il est plus intéressant de voir comment nous en sortir avec sqlpp11. De plus, si on ajoutait des textes l'un après l'autre en exécutant notre programme plusieurs fois, il serait nécessaire de recalculer et mettre à jour le champ **`probabilite`**.

Commençons par déterminer, pour chaque lettre, la somme des nombres d'occurrences de toutes les lettres qui la suivent. En SQL, on écrirait cela plus ou moins ainsi :

Fichier

```
SELECT
   Lettre_Suivante.id_lettre,
   CHAR(Lettres.ucs4 USING utf32) AS lettre,
   SUM(Lettre_Suivante.nombre) AS total
FROM
   Lettre_Suivante
JOIN Lettres ON Lettres.id = Lettre_Suivante.id_lettre
GROUP BY
   Lettre_Suivante.id_lettre;
```

La colonne intermédiaire **`lettre`** (deuxième ligne) n'est là que pour donner une représentation « lisible » de chaque lettre, en transformant le code Unicode de la lettre en un caractère.

L'écriture en sqlpp11 est assez similaire... mais en étant plus tatillonne sur le strict respect de la syntaxe SQL. Remarquez d'abord qu'on utilise un *alias* pour la colonne qui contient la somme : **`AS total`**. Pour pouvoir utiliser un tel alias en C++, il est nécessaire de le déclarer au préalable :

Fichier

```
SQLPP_ALIAS_PROVIDER(total);
```

Du fait de la structure interne du type ainsi déclaré, qui contient en particulier un champ **`static constexpr`**, l'instruction précédente doit être en dehors de la portée d'une fonction.

Par ailleurs, nous allons laisser de côté la transformation du code Unicode en caractère.

On obtient une écriture comme celle-ci :

Fichier

```
01: auto sommes = db(sqlpp::select(
02:    Lettre_Suivante.idLettre,
03:    Lettres.ucs4,
04:    sqlpp::sum(Lettre_Suivante.nombre).as(total)
05:    )
06:    .from(
07:       Lettre_Suivante
08:       .join(Lettres)
09:       .on(Lettres.id == Lettre_Suivante.idLettre)
10:    )
11:    .where(sqlpp::value(true))
12:    .group_by(Lettre_Suivante.idLettre, Lettres.ucs4)
13: );
```

C'est typiquement le genre de syntaxe où les règles de mise en forme et d'écriture du code sont mises à rude épreuve, par le chaînage des appels de fonctions dans les objets retournés.

La clause **`SELECT`** est introduite par la fonction **`sqlpp::select()`**, qui prend en paramètre les colonnes que l'on souhaite... sélectionner. Voyez comme on peut utiliser des fonctions d'agrégation, comme **`sqlpp::sum()`**, ligne 4. On retrouve également à cette ligne l'utilisation de notre *alias*, qui nous permettra par la suite d'accéder aux valeurs de la colonne en question.

L'objet retourné par **`sqlpp::select()`** est nanti d'une fonction **`from()`**. Celle-ci n'attend qu'un seul paramètre, la table dans laquelle on effectue la sélection. N'essayez pas d'écrire **`from(Lettre_Suivante, Lettres)`** : cela ne fonctionne pas. Par contre, cette table peut être le résultat d'une jointure, comme ici : on effectue la jointure entre **`Lettre_Suivante`** et **`Lettres`** (lignes 7 et 8) sur leurs champs respectifs **`id`** et **`idLettre`** (ligne 9). Voyez comme la condition de la jointure est inscrite en utilisant une syntaxe C++ classique. À ce propos, il peut être intéressant de garder à l'esprit que les conjonctions logiques **`and`**, **`or`**, etc. sont des mots réservés en C++, autorisant une écriture plus proche du SQL que l'utilisation des **`&&`**, **`||`** et consorts.

On retrouve la clause **`GROUP BY`** en ligne 12. Mais contrairement à la requête en SQL, on donne également la deuxième colonne – qui est en fait logiquement équivalent à la première, puisqu'il s'agit du caractère associé à l'identifiant. C'est en fait tout simplement le strict respect des règles SQL : lorsqu'une requête est une agrégation, comme ici, alors toutes les colonnes doivent être soit mentionnées dans la clause **`GROUP BY`**, soit passées par une fonction d'agrégation, comme **`SUM()`**, **`AVG()`**, etc. Si cette règle n'est pas respectée, le compilateur exprimera son désaccord, tandis que le moteur MariaDB est plus tolérant.

Enfin, la ligne 11 peut surprendre : pourquoi ajouter ici une clause **`WHERE`**, alors que nous n'avons pas de condition sur le résultat ? À nouveau, cela découle d'une interprétation peut-être un peu pédante de la syntaxe SQL, qui dit qu'une clause **`GROUP BY`** doit « toujours » suivre une clause **`WHERE`**. sqlpp11 exige cela. C'est pourquoi on insère cette clause ici, qui

en fait ne conditionne rien. On aurait pu remplacer `where(sqlpp::value(true))` par `unconditionally()` pour obtenir le même effet et une écriture sans doute plus lisible.

L'utilisation de `auto` pour le type du résultat de l'exécution de la requête (ligne 1) est vivement recommandée. Si vous êtes allergique à `auto` et préférez écrire les types explicitement, voici le « vrai » type de la variable `sommes` :

Fichier

```
sqlpp::result_t<
   sqlpp::mysql::char_result_t,
   sqlpp::result_row_t<
      sqlpp::mysql::connection,
      sqlpp::field_spec_t<
         LS::LettreSuivante_::IdLettre::_alias_t,
         sqlpp::integral, false, false
      >,
      sqlpp::field_spec_t<
         LS::Lettres_::Ucs4::_alias_t,
         sqlpp::integral, true, false
      >,
      sqlpp::field_spec_t<
         total_t::_alias_t,
         sqlpp::integral, true, false
      > > > sommes = db(/*...etc...*/) ;
```

Finalement, `auto` n'est peut-être pas si mal.

7. MISE À JOUR

Maintenant que nous avons nos totaux, nous pouvons mettre à jour les probabilités de succession pour chaque lettre, en itérant sur le résultat de la requête précédente :

Fichier

```
01: for(auto const& somme: sommes)
02: {
03:     double const total = somme.total;
04:     db(sqlpp::update(Lettre_Suivante)
05:        .set(Lettre_Suivante.probabilite =
06:             Lettre_Suivante.nombre / total)
07:        .where(Lettre_Suivante.idLettre == somme.idLettre));
08: }
```

À nouveau, vous ne voulez pas connaître le type exact de la variable `somme`. Vraiment. Mais voyez comme elle nous permet d'accéder aux champs du résultat de la requête : la colonne `total` (ligne 3) et la colonne `idLettre` (ligne 7), qui étaient mentionnées dans les paramètres de la requête `select()`.

Le reste de l'écriture est assez similaire à l'équivalent en pur SQL, en utilisant le chaînage des appels de fonctions. Si vous voulez modifier plusieurs valeurs, listez simplement les affectations données en paramètres de `set()`, par exemple (absurde) :

Fichier

```
05:        .set(Lettre_Suivante.probabilite =
06:             Lettre_Suivante.nombre / total,
06b:            Lettre_Suivante.idLangue = 42)
```

Il aurait sans doute été possible de combiner le **`select()`** et le **`update()`** en une seule instruction, en utilisant une sous-requête, ce que permet sqlpp11. Mais nous aurions obtenu une écriture particulièrement difficile à suivre, des appels chaînés imbriqués les uns dans les autres. Il vaut mieux conserver un équilibre entre une écriture très proche du SQL, mais difficilement lisible du fait de la syntaxe du C++, et une écriture « pur C++ » qui perdrait les bénéfices du moteur de base de données.

8. EXPLOITATION GRAPHIQUE

Nous pouvons maintenant exploiter nos données pour réaliser un diagramme représentatif de la fréquence de succession d'une lettre par une autre. Premier problème : obtenir la liste complète des lettres présentent dans une langue, sachant qu'il est possible qu'une lettre ne soit jamais suivie par aucune autre (par exemple, le *sigma* grec qui s'écrit ς en fin de mot, mais σ ailleurs), ou au contraire qu'elle ne soit jamais précédée par aucune autre (par exemple, le *bêta* grec qui s'écrit β en début de mot, mais ϐ ailleurs, dans certaines traditions d'écriture).

En SQL, on obtient cette liste facilement :

Fichier

```
SELECT DISTINCT Lettres.id
FROM Lettres
JOIN Lettre_Suivante
ON
   (Lettres.id = Lettre_Suivante.id_lettre)
   OR
   (Lettres.id = Lettre_Suivante.id_lettre_suivante)
WHERE
   Lettre_Suivante.id_langue = 1;
```

L'important étant ici la restriction **`DISTINCT`**, pour éviter la répétition. Avec sqlpp11, on écrira :

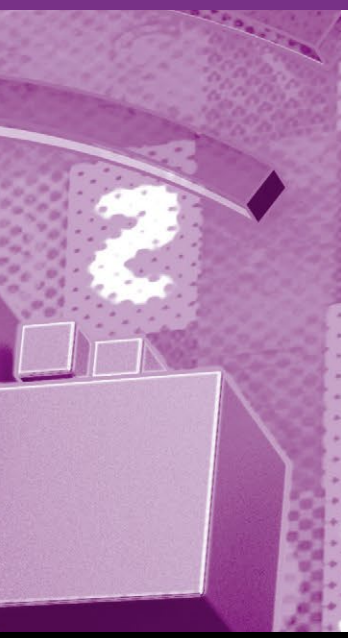
Fichier

```
01: auto les_lettres_sql =
02:    db(sqlpp::select()
03:       .flags(sqlpp::distinct)
04:       .columns(Lettres.id, Lettres.ucs4)
05:       .from(Lettres
06:             .join(Lettre_Suivante)
07:             .on(
08:                (Lettres.id == Lettre_Suivante.idLettre) or
09:                (Lettres.id == Lettre_Suivante.idLettreSuivante)
10:                )
11:            )
12:       .where(Lettre_Suivante.idLangue == id_langue)
13:       );
```

Cette fois, on ne donne plus les colonnes à sélectionner en paramètre de **`select()`**, mais en paramètre d'une fonction **`columns()`** (ligne 4). Cela permet d'insérer un appel à **`flags()`**, par lequel on transmet l'équivalent du **`DISTINCT`**. Une autre façon de l'écrire est :

Fichier

```
auto les_lettres_sql =
   db(sqlpp::select(Lettres.id, Lettres.ucs4)
      .flags(sqlpp::distinct)
      .from(Lettres
// etc.
```



C'est plus court, mais peut-être plus éloigné de la syntaxe SQL. À chacun de trouver son équilibre.

Cela va nous permettre d'ordonner les lettres « comme il faut » : en effet, la seule utilisation du code Unicode (champ **ucs4**) ne donnerait pas le résultat attendu. Par exemple, numériquement, **à** est « après » **z**.

On stocke d'abord la liste des lettres dans un tableau, identifiant et code Unicode :

Fichier

```
std::vector<std::pair<size_t, char32_t>> les_lettres;
for(auto const& l: les_lettres_sql)
{
    les_lettres.push_back({l.id, l.ucs4});
}
```

Difficile de faire plus simple.

Puis on ordonne le tableau, en utilisant par exemple une fonction fournie par Qt **[8]** qui fait cela correctement :

Fichier

```
std::sort(les_lettres.begin(),
          les_lettres.end(),
          [](auto const& p1, auto const& p2)
{
   QString const s1 = QString::fromUcs4(&p1.second, 1);
   QString const s2 = QString::fromUcs4(&p2.second, 1);
   return QString::localeAwareCompare(s1, s2) < 0;
});
```

À noter qu'obtenir les lettres dans le bon ordre est possible en SQL pur, en utilisant **COLLATE**. Malheureusement votre serviteur n'a pas trouvé comment obtenir un équivalent avec sqlpp11.

Il suffit ensuite de composer deux boucles autour de notre tableau de lettres, d'interroger la base de données pour obtenir la probabilité qu'une lettre en suive une autre, et de calculer une couleur à partir d'un dégradé. Schématiquement, cela donne ça :

Fichier

```
01: for(auto const& l_src: les_lettres)
02: {
03:    for(auto const& l_dst: les_lettres)
04:    {
05:       auto proba_sql =
06:          db(sqlpp::select(Lettre_Suivante.probabilite)
07:             .from(Lettre_Suivante)
08:             .where(
09:                (Lettre_Suivante.idLangue == id_langue) and
10:                (Lettre_Suivante.idLettre == l_src.first) and
11:                (Lettre_Suivante.idLettreSuivante == l_dst.first)
12:              )
13:           );
14:       double proba =
15:          proba_sql.empty() ?
16:          0.0 :
17:          proba_sql.front().probabilite;
18:       // calcul savant d'une couleur et dessin...
19:    }
20: }
```

À savoir

Pour ceux qui s'inquièteraient de la construction de multiples `QString` ici, qu'ils se rassurent : chaque chaîne ne contenant qu'un seul caractère (ou deux, car `QString` stocke en UTF16), l'optimisation *small strings optimisation* s'applique et aucune allocation mémoire n'a lieu.

On n'attend au plus qu'une seule valeur de la requête. Plutôt que d'écrire une boucle comme auparavant, on teste si le résultat est vide (ligne 15) avec `empty()`, et sinon on récupère simplement le premier élément (ligne 17) avec `front()`.

Avec les instructions de dessin qui vont bien, cela nous donne la figure 2. Son aspect est différent du tableau proposé dans la vidéo **[5]** du fait, d'une part, d'un algorithme d'analyse plus simpliste, et d'autre part, de l'intégration des lettres accentuées.

AVERTISSEMENT !

Le champ `probabilite` est ici de type `double`. En interne, sqlpp11 utilise `std::strtod()`, dont le fonctionnement dépend de la locale. Si vous n'obtenez que des valeurs entières, c'est probablement que la conversion en `double` depuis la chaîne de caractères renvoyée par l'API « bas niveau » de MariaDB a échoué, par la confusion entre `'.'` et `','` comme séparateur décimal. Ajouter une instruction comme `std::setlocale(LC_NUMERIC, "C");` peut aider.

CONCLUSION

Nous n'avons pas abordé dans cet article toutes les possibilités offertes par sqlpp11, comme les requêtes dynamiques, mais vous devriez néanmoins déjà être en mesure de réaliser vos propres expérimentations. Remarquez que dans tout ce qui précède, à aucun moment nous n'avons explicitement composé une requête SQL dans une chaîne de caractères : tout est encapsulé par sqlpp11, nous laissant avec le seul C++. Tout aussi remarquable, bien que sqlpp11 fasse un usage intensif des *templates* C++, nous n'en avons pas croisé un seul – explicitement du moins.

Vous rencontrerez certaines limitations, et comme l'indique son auteur, sqlpp11 ne supporte pas (encore) tout le standard SQL. Mais cette petite bibliothèque est une alternative très intéressante pour obtenir des programmes plus robustes, en évitant les pièges liés aux manipulations « bas niveau » entre types variants, `char*` et `void*`. Donc si vous êtes un programmeur C++, envisagez de participer au développement. Qui sait, peut-être un jour tout cela constituera les fondements d'une nouvelle section dans le standard C++. ■

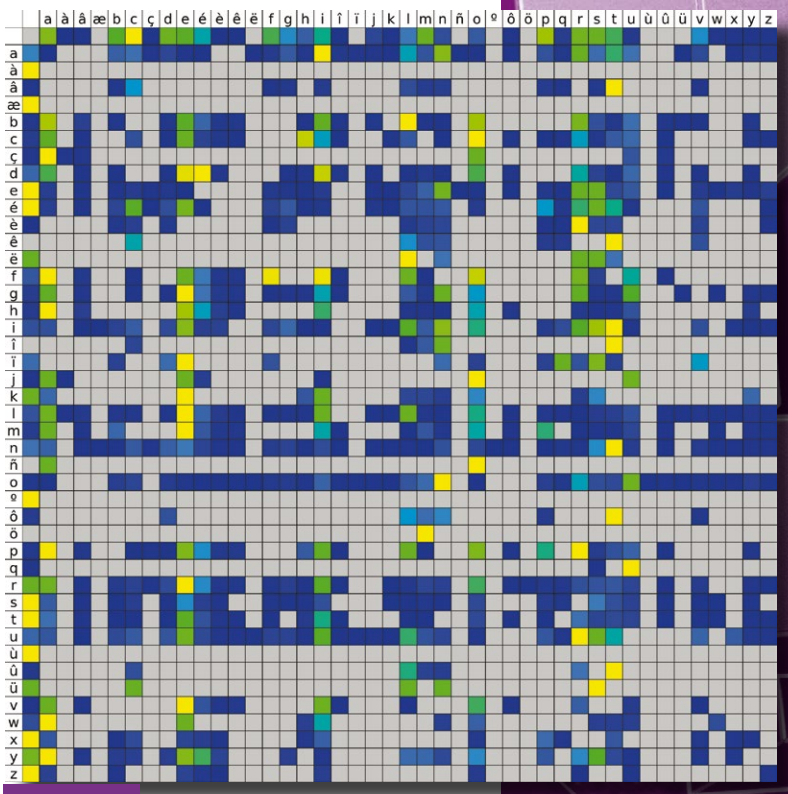


Figure 2

Représentation graphique de la probabilité de succession des lettres en français.

RÉFÉRENCES

[1] sqlpp11 : **https://github.com/rbock/sqlpp11**

[2] Boost : **http://www.boost.org/**

[3] Pyparsing : **http://pyparsing.wikispaces.com/**

[4] CMake : **https://cmake.org/**

[5] David Louapre, « *La machine à inventer des mots* », vidéo et article : **https://sciencetonnante.wordpress.com/2015/10/16/la-machine-a-inventer-des-mots-video/**

[6] David Louapre, « *La machine à inventer des mots, version Ikea* », compléments : **https://sciencetonnante.wordpress.com/2015/11/06/la-machine-a-inventer-des-mots-version-ikea/**

[7] Le projet Gutenberg : **http://www.gutenberg.org/**

[8] QtCore : **http://doc.qt.io/qt-5/qtcore-index.html**

ÉTAPE 5

JE CRÉE DES PROGRAMMES UTILISANT MES BASES

UTILISEZ MYSQL AVEC L'API REST ET PYTHON

Sylvain Nayrolles

De nombreux frameworks web clients permettent de communiquer via une API REST. Nous allons voir comment configurer une base de données relationnelle telle que MySQL pour en faire un backend RESTFul.

Il est de plus en plus fréquent, dans les architectures d'applications web, de rencontrer des modèles reposant sur un *backend* dit RESTFul, c'est-à-dire respectant l'intégralité des contraintes d'une API dite REST. Certains *frameworks* client, tels que **backbones.js** ou bien **Angular.js**, reposent intégralement ou en partie sur une telle architecture. Ces modèles ont essentiellement émergé avec les bases **NoSQL** qui ont popularisé le modèle dit CRUD. Pour éviter de rater le virage, les bases relationnelles commencent à pointer le bout de leur nez, en se reposant essentiellement sur des couches logicielles externes. Mais nous verrons qu'**Oracle** propose un *plugin* pour **MySQL**, dans une version de test, permettant de requêter notre base via une API REST.

1. REST

Le modèle fut introduit par M. Roy Fielding, l'un des pères du HTTP et membre fondateur de la fondation **Apache**. Il est aussi l'un des principaux contributeurs au serveur web portant le même nom. Même si le REST est souvent associé au HTTP, dans sa définition rien ne permet d'en faire l'amalgame, même si le HTTP fût cité par Roy Fielding dans sa thèse définissant ce dernier, comme un support efficace pour mettre en place une API REST. Le REST étant un modèle et non un protocole, il impose des contraintes architecturales :

- une indépendance client-serveur ;
- sans état. Certainement la contrainte principale, car c'est de cette dernière que le modèle tire son nom, car REST signifie *REpresentational State Transfer* ;
- cache : la possibilité de mettre en cache les réponses, ainsi que de définir les critères de mise en cache pouvant être interprétés par des serveurs mandataires intermédiaires ;
- système en couches, permettant de décorréler le client du serveur et pouvoir faire de la redondance et du *load balancing* ;
- *code-on-demand* : contrainte permettant au serveur d'envoyer du code afin de spécialiser le client. Cette contrainte n'est que très rarement prise en compte, car elle impose une gestion d'états, incompatible avec les contraintes précédentes ;
- une interface uniforme.

Le HTTP permet de respecter les contraintes imposées par le modèle, car c'est essentiellement un protocole dit sans état et déconnecté.

De plus, dans sa spécification, le HTTP définit un ensemble de **verbes** pouvant servir une API REST :

- `OPTIONS`
- `GET`
- `HEAD`
- `POST`
- `PUT`
- `DELETE`
- `TRACE`
- `CONNECT`

Certaines spécifications, comme le **WebDav**, définissent une extension de ces verbes. Dans le cadre de la définition d'une API REST rien ne nous l'empêche. Mais beaucoup de matériels de sécurité vont interdire l'utilisation de verbes non standards, voire interdire certains verbes standards.

Nous allons par la suite essayer d'utiliser des verbes standards, car nous allons réaliser une application dite CRUD pour **C**reate, **R**ead, **U**pdate, **D**elete.

Cela regroupe les actions élémentaires que nous pouvons réaliser sur une table de notre base. Il faut bien comprendre qu'une API REST va privilégier les interfaces simples avec la donnée. Ce qui va inévitablement déplacer la complexité du côté du client.

Pour illustrer ce type d'utilisation, nous allons réaliser une simple application de consultation d'articles de *GNU/Linux Magazine*, selon deux alternatives que nous offre MySQL. La première fera intervenir un *middleware* **Python** permettant de simuler le frontal REST HTTP ; la seconde, plus récente dans le monde MySQL et encore en phase de test, sera réalisée grâce à un *plugin* développé par Oracle.

2. MIDDLEWARE

Dans cette section, nous allons utiliser une couche logicielle externe qui réalisera l'interface REST ainsi que le protocole de communication entre le *frontend* et le *backend*. Le but sera ici d'implémenter un modèle CRUD. Le *frontend* pourrait se concrétiser par une simple application web reposant sur Angular, car ce dernier propose une intégration des API REST de façon native. Étant des programmeurs *backend*, nous utiliserons `curl` ! Le *backend* utilisera une base de données MySQL. Il est tout à fait possible, pour cette partie, d'utiliser une base MariaDB. La partie REST utilisera **Flask**, un *framework* web Python côté serveur en combinaison avec l'extension Flask-REST, ainsi que l'aide du module Python `python-mysql`. Pour les utilisateurs MariaDB, le module Python est inchangé.

2.1 MySQL

Nous n'allons rien apprendre de plus que vous n'ayez déjà exploré dans ce hors-série. C'est pour cela que je serais donc concis. Nous allons installer la version 5.7 de MySQL via les dépôts d'Oracle. Nous faisons le choix d'une **Debian**. Il faut tout d'abord télécharger le fichier `mysql-apt-config_0.8.0-1_all.deb` disponible sur le site de téléchargement d'Oracle, sous l'onglet APT **[1]** :

Terminal
```
$ sudo dpkg -i mysql-apt-config_0.8.0-1_all.deb
```

Lors de l'installation, ce dernier vous demandera quelle version de MySQL utiliser ; pour anticiper la prochaine partie, nous allons utiliser la version 5.7.

Il suffit ensuite d'*updater* notre liste de dépôt :

Terminal
```
$ sudo apt-get update
```

Puis de procéder à l'installation :

Terminal
```
$ sudo apt-get install mysql-server
```

Enfin, nous allons peupler notre base en créant une table `articles` et en la peuplant avec quelques informations :

Fichier

```
CREATE DATABASE REST;
USE REST;
CREATE TABLE ARTICLES (
id INT(6) UNSIGNED AUTO_INCREMENT PRIMARY KEY,
titre VARCHAR(30) NOT NULL,
auteur VARCHAR(30) NOT NULL
) ;
INSERT INTO ARTICLES (titre, auteur) VALUES ("MySQL via REST API",
"Sylvain Nayrolles") ;
```

2.2 Flask

Flask est un *framework* web serveur écrit entièrement en Python. Il met en avant sa simplicité et sa modularité. Nous allons nous servir de ce dernier comme *middleware* réalisant la couche REST entre la partie cliente et notre base de données relationnelle, en exposant son API au travers de HTTP.

Pour ce faire, nous allons utiliser une bonne pratique du Python qui consiste à créer un environnement virtuel pour réaliser notre développement :

Terminal

```
$ mkvirtualenv mysql-rest-api --python=$(which python3)
```

Puis nous allons installer tous les modules dont nous avons besoin :

Terminal

```
(mysql-rest-api) > pip install flask flask-restful mysql-connector
```

Nous allons mettre en œuvre notre *middleware* avec très peu de lignes de code. Nous allons essayer de suivre au plus près les coutumes REST observées actuellement sur Internet. Pour cela, nous devons distinguer deux types de requêtes possibles ; les requêtes spécifiant un id ou non. Enfin, nous allons utiliser une symbolique de verbes HTTP afin d'assurer notre périmètre fonctionnel CRUD :

POST	Create
GET	Read
PUT	Update
DELETE	Delete

Notre application Flask minimaliste va définir deux ressources de type REST. De plus, nous allons créer un *pool* de connexions en direction de notre base de données qui nous permettra de réaliser nos requêtes :

Fichier

```
#!/usr/bin/env python3

import textwrap
from flask import Flask
from flask_restful import Api, Resource, abort, reqparse
from mysql.connector.pooling import MySQLConnectionPool

if __name__ == "__main__":
    app = Flask("MySQL via REST API")
    api = Api(app)
```

```
    connection_pool = MySQLConnectionPool(host="localhost", port=8888,
user="foo", password="bar", database="rest")
    api.add_resource(ArticlesList, "/crud/articles", resource_class_
kwargs={"connection_pool": connection_pool})
    api.add_resource(Articles, "/crud/articles/<article_id>", resource_class_
kwargs={"connection_pool": connection_pool})
    app.run(debug=True)
```

Les deux premières lignes permettent de créer notre application au sens Flask. Ensuite, nous créons notre *pool* de connexion avec la configuration appropriée afin de réaliser les opérations nécessaires. Enfin, nous définissons deux ressources :

- ⇨ **`/crud/articles`** afin de manipuler la table **`article`** dans son ensemble (lister et insérer) sans notion d'identifiant ;
- ⇨ **`/crud/articles/<article_id>`** pour manipuler les articles en les identifiant par leur ID (mise à jour, suppression, lecture).

Chaque ressource est typée par une classe que nous allons définir. **`/crud/articles`** est typé par la vue **`ArticlesList`** :

Fichier

```
class ArticlesList(Resource):
    def __init__(self, *, connection_pool, **kwargs):
        super().__init__(**kwargs)
        self._connection_pool = connection_pool

    def get(self):
        connection = self._connection_pool.get_connection()
        cursor = connection.cursor()
        cursor.execute(textwrap.dedent("""
            SELECT id, titre, auteur
            FROM article;
            """))
        rows = cursor.fetchall()
        cursor.close()
        connection.close()

        return [{
            "id": row[0],
            "titre": row[1],
            "auteur": row[2]
        } for row in rows]

    def post(self):
        parser = reqparse.RequestParser()
        parser.add_argument("titre")
        parser.add_argument("auteur")
        args = parser.parse_args(strict=True)

        connection = self._connection_pool.get_connection()
        cursor = connection.cursor()
        cursor.execute(textwrap.dedent("""
            INSERT INTO articles (titre, auteur)
            VALUES ("%(titre)s", "%(auteur)s");
            """ % (args)))
        new_id = cursor.lastrowid
        cursor.close()
        connection.commit()
        connection.close()
        return Articles(connection_pool=self._connection_pool).get(new_id), 201
```

La fonction **`get`** permet de récupérer l'ensemble des articles présents dans la base de données. La fonction **`post`** permet de récupérer, de créer une nouvelle entrée dans la table. Le **`reqparse`** nous permet de nous assurer de la cohérence des données passées en paramètres. Nous observons que nous retenons le code HTTP **`201`** dans le cadre de la création.

À PROPOS DES KEYWORD-ONLY ARGUMENTS

Le PEP 3102 (**https://www.python.org/dev/peps/pep-3102/**) définit les « keyword-only arguments » : comment déclarer une fonction en spécifiant que certains paramètres ne peuvent être utilisés qu'en les nommant lors de l'appel. Pour séparer les paramètres de position des paramètres « keyword-only », on utilise le caractère *****.

Déclarons par exemple la fonction suivante :

Fichier

```
def fct(a, b, *, key=5) :
    print(a, b, key)
```

Il est possible d'appeler la fonction par **`fct(a, b)`** ou **`fct(a, b, key=4)`**.

On aurait pu la déclarer différemment :

Fichier

```
def fct(a, b, key=5) :
    print(a, b, key)
```

Mais ici il aurait été possible d'appeler **`fct(a, b, 4)`**…

`/crud/articles/<article_id>` est typé par la vue **`Articles`** :

Fichier

```
class Articles(Resource):
    def __init__(self, *, connection_pool, **kwargs):
        super().__init__(**kwargs)
        self._connection_pool = connection_pool

    def get(self, article_id):
        connection = self._connection_pool.get_connection()
        cursor = connection.cursor()
        cursor.execute("SELECT id, titre, auteur FROM articles WHERE id=%s" % article_id)
        row = cursor.fetchone()
        cursor.close()
        connection.close()

        if row is None:
            abort(404, message="Unknown article %s" % article_id)

        return {
            "id": row[0],
            "titre": row[1],
            "auteur": row[2]
        }

    def delete(self, article_id):
        connection = self._connection_pool.get_connection()
        cursor = connection.cursor()
        cursor.execute("DELETE FROM articles WHERE id = '%s'" % article_id)
        cursor.close()
        connection.commit()
        connection.close()
        return '', 204

    def put(self, article_id):
        parser = reqparse.RequestParser()
        parser.add_argument("titre")
```

```
        parser.add_argument("auteur")
        args = parser.parse_args(strict=True)

        connection = self._connection_pool.get_connection()
        cursor = connection.cursor()
        cursor.execute(textwrap.dedent("""
            UPDATE article
            SET titre="%s",auteur="%s"
            WHERE id=%s;
            """ % (args["titre"], args["auteur"], article_id)))
        cursor.close()
        connection.commit()
        connection.close()
        return Articles(connection_pool=self._connection_pool).get(article_id), 201
```

La fonction **get** permet donc de retrouver un article en particulier, tandis que la fonction **delete** permet de supprimer une entrée. Enfin, la fonction **put** permet de mettre à jour notre ligne dans la table via la fonction **UPDATE** de MySQL.

Nous pouvons utiliser **curl** afin de tester notre serveur REST. Nous allons afficher tous les articles présents dans la base :

Terminal

```
$ curl http://localhost:5000/crud/articles
[
    {
        "auteur": "Sylvain Nayrolles",
        "id": 1,
        "titre": "MySQL via REST API"
    }
]
```

Si nous voulons juste l'article avec l'id **1** :

Terminal

```
$ curl http://localhost:5000/crud/articles/1
{
    "auteur": "Sylvain Nayrolles",
    "id": 1,
    "titre": "MySQL via REST API"
}
```

Afin de créer une nouvelle entrée dans notre base de données, nous allons utiliser le verbe **POST** :

Terminal

```
$ curl http://localhost:5000/crud/articles -X POST -d "titre=foo&auteur=bar"
{
    "auteur": "bar",
    "id": 2,
    "titre": "foo"
}
```

Zut, une faute de typo :

Terminal

```
$ curl http://localhost:5000/crud/articles/2 -X PUT -d
"titre=foo&auteur=bistrot"
{
    "auteur": "bistrot",
    "id": 2,
    "titre": "foo"
}
```

En fait cet article n'existe pas, je vais donc le supprimer en utilisant le verbe **DELETE** :

Terminal
```
$ curl http://localhost:5000/crud/articles/7 -X DELETE
```

Voici notre API prête et RESTFull. L'inconvénient, c'est que l'ajout d'un *middleware* peut vite représenter un nœud de performance dans notre application. Nous allons voir comment, dans un futur proche, utiliser le *plugin* natif à MySQL.

3. MYSQL HTTP PLUGIN

Il existe donc un *plugin* MySQL développé par Oracle offrant à ladite base de données une API REST transportée par du HTTP et dont les réponses sont formatées en JSON. D'ailleurs ce *plugin* se nomme MySQL HTTP *plugin* et il est disponible en téléchargement sur le site de MySQL [2]. Le téléchargement englobe une version complète de MySQL 5.7 ; il n'est donc pas possible à l'heure actuelle de patcher une version officielle sans bidouille, car le *plugin* est encore en phase expérimentale. Pour l'installer, il suffit de suivre la procédure présente au sein du fichier **INSTALL-BINARY**. Si vous rencontrez des problèmes lors de l'installation, c'est peut-être qu'il vous manque une dépendance non standard dans une distribution de type Debian :

Terminal
```
$ sudo apt-get install libaio1
```

Une fois le service lancé, il ne nous reste plus qu'à charger le module depuis une invite de commandes mysql:

Terminal
```
mysql> INSTALL PLUGIN myhttp SONAME 'libmyhttp.so';
```

Dans sa configuration par défaut, ce module nous offre deux modes :

- un mode dans lequel il est possible de réaliser des requêtes SQL directement ;
- un mode dit CRUD que maintenant vous maîtrisez parfaitement.

Nous allons donc tester le second mode qui fonctionne exactement comme le module *middleware* développé dans la section précédente.

Nous allons tout d'abord configurer notre module pour qu'il pointe vers notre base de données. Pour ce faire, nous allons éditer le fichier de configuration de MySQL, souvent situé dans **/etc/my.cnf** lors d'une installation de test :

Fichier
```
[mysqld]
myhttp_default_db = rest
myhttp_default_mysql_user_name = foo
myhttp_default_mysql_user_passwd = bar
myhttp_default_mysql_user_host = 127.0.0.1
```

À retenir

Dans cet exemple, nous n'avons pas géré l'authentification. L'authentification est une question épineuse dans une API REST, car elle casse le modèle, et elle nécessite de conserver un état. Plusieurs solutions s'offrent à nous, mais aucune n'est réellement satisfaisante. Une bonne pratique permet soit d'utiliser l'authentification TLS, soit de gérer des clés d'API via l'implémentation du protocole OAuth2. Il existe une extension de Flask permettant de gérer ce dernier.

Une fois le serveur redémarré, afin de prendre en compte notre nouvelle configuration, nous allons la tester toujours via notre bon `curl` :

Terminal

```
$ curl --user basic_auth_user:basic_auth_passwd --url http://127.0.0.1:8080/
crud/rest/articles/1
{"id":"1","titre":"MySQL via REST API","auteur":"Sylvain Nayrolles"}
```

Nous notons que l'authentification est réalisée en HTTP basic avec les identifiants `basic_auth_user`, et `basic_auth_passwd`.

Pour l'ajout, à la différence de la section précédente, les informations sont formatées en JSON également :

Terminal

```
$ curl --user basic_auth_user:basic_auth_passwd --url http://127.0.0.1:8080/
crud/rest/articles/2 -d '{"titre":"foo", "auteur":"bar"}' -H "Accept:
application/json" -X PUT -v
{"affected_rows":1,"warning_count":0}
```

Et tout simplement pour supprimer :

Terminal

```
$ curl --user basic_auth_user:basic_auth_passwd --url http://127.0.0.1:8080/
crud/rest/articles/2 -X DELETE -v
```

Un *plugin* vraiment intéressant dans son fonctionnel, mais encore en phase de validation chez Oracle, qui nous laisse entrevoir la volonté de ce dernier de faire de sa base de données un moteur indispensable à toute application web moderne, car avec ce *plugin* il est donc possible d'envisager la rigueur du monde relationnel allié à la souplesse d'une API REST.

CONCLUSION

Au travers de cet article, nous avons pu constater que la mort annoncée du modèle relationnel au profit des bases NoSQL est loin d'être faite. Avec ce genre d'initiatives, à mon sens, les bases relationnelles reprennent une longueur d'avance grâce à la rigueur de leur modèle. Une dernière analyse aurait pu être les performances de ce type d'architectures alliées à une base de données telle que MySQL. Le *plugin* natif permet de considérablement améliorer ce type de statistiques ; mais qu'en est-il de la sécurité ? ■

RÉFÉRENCES

[1] Fichier de configuration du dépôt MySQL:
http://dev.mysql.com/downloads/repo/apt/

[2] Site de téléchargement du *plugin* HTTP pour MySQL :
https://labs.mysql.com/

DÉCOUVREZ LES NOUVELLES FONCTIONS NATIVES SQL POUR MANIPULER DU CONTENU JSON

Christian Soutou

Depuis la version 5.7.12 de MySQL, le terme « Document store » préfigure des fonctionnalités NoSQL. La gestion de données non structurées constitue pour l'instant une première avancée. Cet article présente les nouvelles fonctions natives SQL pour manipuler du contenu JSON.

Jusqu'en version 5.7, **MySQL** n'était pas particulièrement adapté à la gestion de données non structurées. Les fonctions proposées pour la gestion de **XML** sont toujours très pauvres et n'ont pas évolué depuis bon nombre d'années. Depuis les dernières *releases* de la version 5.7, l'accent a été mis sur la gestion de contenu JSON [1], [2]. Concernant **SQL**, de nouvelles fonctions sont apparues pour :

- générer du contenu (**JSON_ARRAY**, **JSON_OBJECT** et **JSON_QUOTE**) ;
- extraire du contenu (**JSON_CONTAINS**, **JSON_CONTAINS_PATH**, **JSON_EXTRACT** et **JSON_KEYS**) ;
- modifier du contenu (**JSON_UNQUOTE**, **JSON_ARRAY_APPEND**, **JSON_ARRAY_INSERT**, **JSON_INSERT**, **JSON_MERGE**, **JSON_REMOVE** et **JSON_REPLACE**) ;
- extraire certaines caractéristiques du contenu (**JSON_DEPTH**, **JSON_LENGTH**, **JSON_TYPE** et **JSON_VALID**).

1. CRÉATION DE LA TABLE

Une colonne de type **JSON** ne peut pas avoir de valeur par défaut et la taille maximale d'un contenu est régie par le paramètre **max_allowed_packet** (de 4Mio à 1Gio suivant les configurations).

Fichier

```
CREATE TABLE tab_vol_json
(id          SMALLINT AUTO_INCREMENT PRIMARY KEY,
 utilisateur VARCHAR(30),
 doc         JSON,
 doc_chaine  VARCHAR(500),
 modif       DATETIME) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

Tout contenu est vérifié dès l'insertion ou après modification. Il est donc impossible de stocker un document invalide. Dans l'exemple suivant, il manque une double quote au numéro du vol. Cette information qui n'est pas valide, pourrait être insérée dans la colonne **doc_chaine** de type **VARCHAR** :

Fichier

```
mysql> INSERT INTO tab_vol_json
  ->   (utilisateur, doc, modif)
  ->   VALUES ('Brouard','{"date_vol":"2016-07-18","num_vol":AF6140"}', SYSDATE());
ERROR 3140 (22032): Invalid JSON text: "Invalid value." at position 35 in value for
column 'tab_vol_json.doc'.
```

2. INSERTION DE DOCUMENTS

Le script suivant présente deux insertions correctes :

Fichier

```
mysql> -- dans du VARCHAR le contenu JSON invalide ne pose pas de problème
mysql> INSERT INTO tab_vol_json
    -> (utilisateur, doc_chaine, modif) VALUES
    ->  ('Brouard', '{"date_vol":"2016-07-18","num_vol":AF6140"}', SYSDATE());
Query OK, 1 row affected (0.00 sec)
```

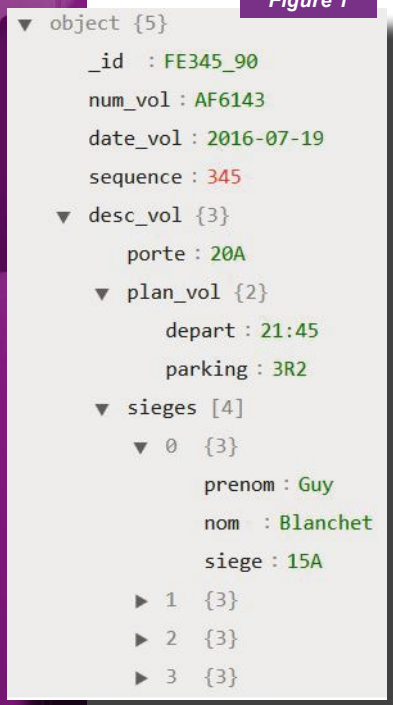
```
mysql> -- dans du VARCHAR le contenu JSON invalide ne pose pas de problème
mysql> INSERT INTO tab_vol_json
    -> (utilisateur, doc_chaine, modif) VALUES
    ->  ('Brouard', '{"date_vol":"2016-07-18","num_vol":AF6140"}', SYSDATE());
Query OK, 1 row affected (0.00 sec)

mysql> -- Insertion d'un document JSON valide dans une colonne JSON
mysql> INSERT INTO tab_vol_json
    -> (utilisateur, doc, modif) VALUES
    ->  ('Brouard','{"date_vol":"2016-07-18","num_vol":"AF6141"}', SYSDATE());
Query OK, 1 row affected (0.18 sec)
```

La structure complète JSON inclut 5 champs au premier niveau : **_id**, **num_vol**, **date_vol**, **sequence** et **desc_vol**. Le dernier champ **desc_vol** est lui-même composé de 5 champs (**porte**, **plan_vol**, **depart**, **parking** et **sieges** qui est un *array* composé d'éléments de 3 champs terminaux : **prenom**, **nom** et **siege**).

L'insertion suivante ajoute le document à la table, deux autres lignes sont ajoutées à la table pour fournir des résultats aux requêtes par la suite.

Figure 1



Structure JSON utilisée pour les exemples à suivre.

Fichier

```
mysql> INSERT INTO tab_vol_json
    -> (utilisateur, doc, modif) VALUES
    ->  ('Salais',
    ->   '{"_id" : "FE345_90","num_vol" : "AF6143",
"date_vol" : "2016-07-19", "sequence" : 345,
    '>    "desc_vol" :
    '>    {"porte" : "20A",
    '>     "plan_vol" : { "depart" : "21:45", "parking"
: "3R2"},
    '>     "sieges" :  [{"prenom" : "Guy",    "nom" :
"Blanchet", "siege" : "15A"},
    '>                 {"prenom" : "Gerard", "nom" :
"Diffis",   "siege" : "15B"},
    '>                 {"prenom" : "Victor", "nom" :
"Ferrage",  "siege" : "15C"},
    '>                 {"prenom" : "Henri",  "nom" :
"Alquie",   "siege" : "13C"}]}
    '>    }', DATE_ADD(SYSDATE(),INTERVAL 1 DAY));
Query OK, 1 row affected (0.03 sec)
```

3. GÉNÉRATION DE CONTENUS

Le script suivant présente quelques contenus JSON générés. La fonction **JSON_ARRAY([expr[, expr]...])** construit un tableau. La fonction **JSON_OBJECT([champ, expr[, champ, expr]...])** construit un document avec des champs. La fonction **JSON_QUOTE(doc_json)** encadre un contenu entre double quotes en préservant éventuellement des quotes internes.

Fichier

```
mysql> SELECT JSON_ARRAY(id, utilisateur, modif)
    -> FROM   tab_vol_json WHERE id BETWEEN 2 AND 4;
+-------------------------------------------+
| JSON_ARRAY(id, utilisateur, modif)        |
+-------------------------------------------+
| [2, "Brouard", "2016-10-09 12:42:58.000000"] |
| [3, "Salais", "2016-10-10 12:42:58.000000"]  |
| [4, "Bizoi", "2016-10-10 12:42:58.000000"]   |
+-------------------------------------------+
mysql> SELECT JSON_OBJECT('num_v',  id,
    ->                    'compte', JSON_ARRAY(utilisateur, DATE(modif)))
    -> FROM  tab_vol_json WHERE utilisateur <> 'Brouard';
+-----------------------------------------------------------------------+
| JSON_OBJECT('num_v',id,'compte',JSON_ARRAY(utilisateur,DATE(modif))) |
+-----------------------------------------------------------------------+
| {"num_v": 3, "compte": ["Salais", "2016-10-10"]}                      |
| {"num_v": 4, "compte": ["Bizoi", "2016-10-10"]}                       |
| {"num_v": 5, "compte": ["Salais", "2016-10-11"]}                      |
+-----------------------------------------------------------------------+
mysql> SELECT id, JSON_QUOTE(utilisateur)
    -> FROM   tab_vol_json WHERE id IN (1,3,4);
+----+--------------------------+
| id | JSON_QUOTE(utilisateur) |
+----+--------------------------+
|  1 | "Brouard"                |
|  3 | "Salais"                 |
|  4 | "Bizoi"                  |
+----+--------------------------+
```

4. EXTRACTION DE CONTENUS

Les scripts suivants présentent les fonctions pour parcourir et extraire du contenu JSON. La fonction **`JSON_CONTAINS(doc_json, expr[, chemin])`** retourne **`1`** si l'expression est trouvée dans le chemin à l'intérieur du document, **`0`** sinon. Notez l'utilisation des doubles quotes pour comparer une chaîne de caractères et le caractère **`$`** qui désigne la racine. La première requête ne trouve pas la valeur **`348`** pour le champ **`sequence`**. La deuxième requête extrait les lignes qui vérifient que le champ **`porte`** soit égal à **`20A`** (notez la notation pointée pour parcourir la structure) ou que le champ **`parking`** ne soit pas renseigné.

Fichier

```
mysql> SELECT JSON_CONTAINS(doc, '"AF6143"' , '$.num_vol') AS contains_AF6143,
    ->        JSON_CONTAINS(doc, '348'      , '$.sequence')     AS contains_
seq_348
    -> FROM   tab_vol_json
    -> WHERE  utilisateur = 'Salais' AND id = 3;
+-----------------+------------------+
| contains_AF6143 | contains_seq_348 |
+-----------------+------------------+
|               1 |                0 |
+-----------------+------------------+
mysql> SELECT id, utilisateur, DATE(modif)
    -> FROM   tab_vol_json
    -> WHERE  JSON_CONTAINS(doc, '"20A"', '$.desc_vol.porte') = 1
```

```
    -> OR      JSON_CONTAINS(doc, 'null' , '$.desc_vol.plan_vol.
parking')=1
    -> ORDER BY id;
+----+-------------+-------------+
| id | utilisateur | DATE(modif) |
+----+-------------+-------------+
|  3 | Salais      | 2016-10-10  |
|  4 | Bizoi       | 2016-10-10  |
|  5 | Salais      | 2016-10-11  |
+----+-------------+-------------+
```

La fonction **JSON_CONTAINS_PATH(doc_json, {'one'|'all'}, chemin[, chemin]...)** retourne **1** si le ou les chemins existent à l'intérieur du document, **0** sinon. La troisième requête cherche les documents ne disposant pas du champ **desc_vol**. La dernière s'intéresse à ceux qui disposent d'un champ **porte** (il n'est pas dit qu'il ne puisse pas être **null**), et d'un quatrième passager.

Fichier

```
mysql> SELECT id, utilisateur, DATE(modif)
    -> FROM   tab_vol_json
    -> WHERE  JSON_CONTAINS_PATH(doc, 'one', '$.desc_vol') = 0;
+----+-------------+-------------+
| id | utilisateur | DATE(modif) |
+----+-------------+-------------+
|  2 | Brouard     | 2016-10-09  |
+----+-------------+-------------+
mysql> SELECT id, utilisateur, DATE(modif)
    -> FROM   tab_vol_json
    -> WHERE  JSON_CONTAINS_PATH(doc, 'all' , '$.desc_vol.porte',
    ->                                        '$.desc_vol.sieges[3].
siege') = 1;
+----+-------------+-------------+
| id | utilisateur | DATE(modif) |
+----+-------------+-------------+
|  3 | Salais      | 2016-10-10  |
+----+-------------+-------------+
```

La fonction **JSON_EXTRACT(doc_json, chemin [,chemin] ...)** extrait le contenu désigné dans le ou les chemins. Depuis la version 5.7.9, l'opérateur **->** joue le même rôle si seulement un chemin est décrit. Les deux premières requêtes extraient le numéro et la date du vol. La première parcourt tous les éléments du tableau **sieges** tandis que la deuxième extrait le champ **porte** sous le champ **desc_vol**.

Fichier

```
mysql> SELECT JSON_EXTRACT(doc, '$.num_vol', '$.date_vol')    AS elements_vol,
    ->        JSON_EXTRACT(doc, '$.desc_vol.sieges[*].siege') AS passagers
    -> FROM   tab_vol_json
    -> WHERE  utilisateur = 'Salais';
+--------------------------+------------------------------+
| elements_vol             | passagers                    |
+--------------------------+------------------------------+
| ["AF6143", "2016-07-19"] | ["15A", "15B", "15C", "13C"] |
| ["AF6143", "2016-07-20"] | ["1A", "1B", "1C"]           |
+--------------------------+------------------------------+
mysql> SELECT id, CONCAT(doc->'$.num_vol', doc->'$.date_vol') AS elt_vol,
    ->        doc->'$.desc_vol.porte' AS porte
    -> FROM   tab_vol_json
    -> WHERE  utilisateur='Salais';
```

```
+----+----------------------+-------+
| id | elt_vol              | porte |
+----+----------------------+-------+
|  3 | "AF6143""2016-07-19" | "20A" |
|  5 | "AF6143""2016-07-20" | "20B" |
+----+----------------------+-------+
```

La fonction **JSON_KEYS(doc_json [, chemin])** liste les champs qui sont présents au niveau du chemin précisé sous la forme d'un tableau. La dernière requête permet de constater que la première ligne est associée à un document absent. Le deuxième document ne dispose que des champs concernant le numéro et la date du vol. Enfin le troisième document est le plus complet en terme de structure.

Fichier

```
mysql> SELECT id, JSON_KEYS(doc)                        AS sous_doc,
    ->            JSON_KEYS(doc, '$.desc_vol.plan_vol') AS sous_plan_vol
    -> FROM   tab_vol_json
    -> WHERE  id IN (1,2,4);
+----+-----------------------------------------------------+---------------------+
| id | sous_doc                                            | sous_plan_vol       |
+----+-----------------------------------------------------+---------------------+
|  1 | NULL                                                | NULL                |
|  2 | ["num_vol", "date_vol"]                             | NULL                |
|  4 | ["_id", "num_vol", "date_vol", "desc_vol", "sequence"]| ["depart", "parking"] |
+----+-----------------------------------------------------+---------------------+
```

5. RETYPAGE DE CONTENUS

La fonction **JSON_UNQUOTE(doc_json)** supprime les double quotes qui encadrent le contenu. En combinant cette fonction avec **CAST,** il est possible de choisir le type de données résultat avant un traitement éventuel (dans l'exemple suivant, une chaîne, une date et un numérique).

Fichier

```
mysql> SELECT id, JSON_UNQUOTE(doc->'$.num_vol')                 AS num_vol,
    ->            JSON_UNQUOTE(doc->'$.date_vol')                AS date_vol,
    ->            JSON_UNQUOTE(doc->'$.desc_vol.sieges[0].siege') AS siege_0
    -> FROM   tab_vol_json
    -> WHERE  utilisateur = 'Salais';
+----+---------+------------+---------+
| id | num_vol | date_vol   | siege_0 |
+----+---------+------------+---------+
|  3 | AF6143  | 2016-07-19 | 15A     |
|  5 | AF6143  | 2016-07-20 | 1A      |
+----+---------+------------+---------+
mysql> SELECT id, CAST(JSON_UNQUOTE(doc->'$.num_vol')  AS CHAR(6))          AS num_vol,
    ->            CAST(JSON_UNQUOTE(doc->'$.date_vol') AS DATE)            AS date_vol,
    ->            CAST(JSON_UNQUOTE(doc->'$.sequence') AS UNSIGNED INTEGER) AS sequence
    -> FROM   tab_vol_json
    -> WHERE  utilisateur = 'Salais';
+----+---------+------------+----------+
| id | num_vol | date_vol   | sequence |
+----+---------+------------+----------+
|  3 | AF6143  | 2016-07-19 |      345 |
|  5 | AF6143  | 2016-07-20 |      347 |
+----+---------+------------+----------+
```

6. MODIFICATION DE CONTENUS

Plusieurs fonctions existent, en premier lieu, la fonction **`JSON_INSERT(doc_json, chemin, expr[,chemin, expr]...)`** permet d'insérer des champs à différents niveaux du contenu qui passe en paramètre. La requête suivante ajoute deux champs (**`sequence`** et **`desc_vol`** lui-même composé) à un contenu initial composé de deux champs (numéro et date du vol).

Fichier

```
mysql> SELECT JSON_INSERT(doc, '$.sequence',344, '$.desc_vol',
    ->                 JSON_OBJECT('porte','14K', 'plan_vol',
    ->                             JSON_OBJECT('depart', null,
'parking','3R4'))) AS json_insert
    -> FROM   tab_vol_json WHERE id = 2;
+-------------------------------------------------------------------------------+
| json_insert                                                                   |
+-------------------------------------------------------------------------------+
| {"num_vol": "AF6141", "date_vol": "2016-07-18", "desc_vol": {"porte":
"14K", "plan_vol":                                                             |
    {"depart": null, "parking": "3R4"}}, "sequence": 344}                     |
+-------------------------------------------------------------------------------+
```

La fonction **`JSON_ARRAY_APPEND(doc_json, chemin, expr[,chemin, expr]...)`** ajoute un élément après le dernier indice d'un tableau (en précisant éventuellement un niveau si des tableaux sont imbriqués). La requête suivante ajoute un passager au tableau **`sieges`** du vol numéro **`3`**.

Fichier

```
mysql> SELECT JSON_ARRAY_APPEND(
    ->         JSON_EXTRACT(doc,'$.desc_vol.sieges'), '$',
    ->            JSON_OBJECT('prenom','Antoine','nom','Derouin',
'siege','5A')) AS json_array_append
    -> FROM   tab_vol_json WHERE id = 3;
+-------------------------------------- ------------------------------+
| json_array_append                                                    |
+----------------------------------------------------------------------+
| [{"nom": "Blanchet", "siege": "15A", "prenom": "Guy"}, {"nom": "Diffis",
"siege": "15B",                                                      |
|  "prenom": "Gerard"}, {"nom": "Ferrage", "siege": "15C", "prenom":
"Victor"}, {"nom":                                                   |
|  "Alquie", "siege": "13C", "prenom": "Henri"}, {"nom": "Derouin","siege":
"5A", "prenom":                                                      |
|  "Antoine"}]                                                         |
+----------------------------------------------------------------------+
```

Doté des mêmes paramètres, la fonction **`JSON_ARRAY_INSERT`** ajoute un élément à un indice choisi d'un tableau. La requête suivante ajoute le même passager au premier indice du tableau.

Fichier

```
mysql> SELECT JSON_ARRAY_INSERT(
    ->         JSON_EXTRACT(doc,'$.desc_vol.sieges'), '$[0]',
    ->            JSON_OBJECT('prenom','Antoine','nom','Derouin',
'siege','5A'))
    ->         AS json_array_insert
    -> FROM   tab_vol_json WHERE id = 3;
+-------------------------------------------------------------------------------+
```

```
| json_array_insert                                                                |
+----------------------------------------------------------------------------------+
| [{"nom": "Derouin", "siege": "5A", "prenom": "Antoine"}, {"nom": "Blanchet",
"siege": "15A",                                                                   |
    "prenom": "Guy"}, {"nom": "Diffis", "siege": "15B", "prenom": "Gerard"},
{"nom":                                                                           |
    "Ferrage", "siege": "15C", "prenom": "Victor"}, {"nom": "Alquie" "siege":
"13C", "prenom":                                                                  |
    "Henri"}]                                                                     |
+----------------------------------------------------------------------------------+
```

Pour remplacer effectivement un fragment dans la table, il suffira d'utiliser une simple instruction **UPDATE** (voir plus loin).

Pour fusionner plusieurs contenus, il existe la fonction **JSON_MERGE(doc_json, [,doc_json]...)**. Pour enlever un fragment un contenu, il faudra utiliser la fonction **JSON_REMOVE(doc_json, chemin [,chemin]...)**. La première requête du script suivant fusionne deux contenus alors que la deuxième supprime les champs **desc_vol** et **sieges** au contenu.

Fichier

```
mysql> SELECT JSON_MERGE((JSON_OBJECT('num_v', id)), doc) AS json_merge
    -> FROM   tab_vol_json WHERE id = 2;
+-------------------------------------------------------------+
| json_merge                                                  |
+-------------------------------------------------------------+
| {"num_v": 2, "num_vol": "AF6141", "date_vol": "2016-07-18"} |
+-------------------------------------------------------------+
mysql> SELECT JSON_REMOVE(doc, '$.sieges', '$.desc_vol') as json_remove
    -> FROM   tab_vol_json WHERE id = 2;
+-------------------------------------------------+
| json_remove                                     |
+-------------------------------------------------+
| {"num_vol": "AF6141", "date_vol": "2016-07-18"} |
+-------------------------------------------------+
```

Par ailleurs, les fonctions **JSON_SET(doc_json, chemin, expr[,chemin, expr]...)** et **JSON_REPLACE** (mêmes paramètres) remplacent des fragments de contenus. La méthode **JSON_SET** a la capacité d'ajouter de nouveaux fragments sans qu'ils existent au préalable alors que **JSON_INSERT** insère un fragment sans remplacer aucun champ existant. La première requête du script suivant modifie deux champs en ajoutant un nouveau champ au contenu. La dernière remplace simultanément trois champs au document.

Fichier

```
mysql> SELECT JSON_SET(doc,
    ->          '$._id',       'FG00_02',
    ->         '$.date_vol', '2016-07-23',
    ->         '$.num_vol' , 'AF5661') AS json_set
    -> FROM   tab_vol_json WHERE id = 2;
+-------------------------------------------------------------------+
| json_set                                                          |
+-------------------------------------------------------------------+
| {"_id": "FG00_02", "num_vol": "AF5661", "date_vol": "2016-07-23"} |
+-------------------------------------------------------------------+
mysql> SELECT JSON_REPLACE(doc
    ->                   '$._id',                    'FG450_92',
    ->                   '$.desc_vol.porte',         '20F',
```

```
    ->                   '$.desc_vol.sieges[0].siege', '12G') AS json_
replace
    -> FROM   tab_vol_json WHERE utilisateur = 'Bizoi';
+--------------------------------------------------------------------------+
| json_replace                                                             |
+--------------------------------------------------------------------------+
| {"_id": "FG450_92", "num_vol": "AF6146", "date_vol": "2016-07-19",
"desc_vol": {"porte":                                                     |
|  "20F", "sieges": [{"nom": "Blanchet", "siege": "12G", "prenom": "Guy"},
{"nom": "Diffis",                                                         |
|  "siege": "3B", "prenom": "Gerard"}], "plan_vol": {"depart": "21:55",
"parking": "3R6"}},                                                       |
| "sequence": 346}                                                         |
+--------------------------------------------------------------------------+
```

7. EXTRACTION DE CARACTÉRISTIQUES

Il existe des fonctions qui renseignent à propos des caractéristiques de documents JSON stockés. La fonction **`JSON_DEPTH(doc_json)`** retourne la « profondeur » maximum du document (**`1`** pour un scalaire, **`2`** pour un objet qui contient un champ ou pour un tableau qui contient un scalaire, etc.). Dans le même ordre d'idée, la fonction **`JSON_LENGTH(doc_json[, chemin])`** retourne la taille du document en considérant d'abord le nombre d'objets membres et en considérant la taille d'un tableau comme le nombre d'éléments de celui-ci.

La première requête du script suivant renseigne à propos de différents niveaux des documents sélectionnés. La deuxième requête présente deux tailles : celle du document dans sa globalité (nombre de champs du premier niveau) et celle du tableau **`sieges`** (nombre d'éléments dans chaque collection).

Fichier

```
mysql> SELECT id, utilisateur, JSON_DEPTH(doc) AS profondeur_maxi
    -> FROM   tab_vol_json
    -> WHERE  utilisateur IN ('Bizoi', 'Brouard')
    -> ORDER BY JSON_DEPTH(doc) DESC;
+----+-------------+----------------+
| id | utilisateur | profondeur_maxi |
+----+-------------+----------------+
|  4 | Bizoi       |              5 |
|  2 | Brouard     |              2 |
|  1 | Brouard     |           NULL |
+----+-------------+----------------+
mysql> SELECT id, JSON_LENGTH(doc)                        AS taille,
    ->            JSON_LENGTH(doc, '$.desc_vol.sieges') AS taille_
sieges
    -> FROM   tab_vol_json WHERE id IN (1,2,3);
+----+--------+---------------+
| id | taille | taille_sieges |
+----+--------+---------------+
|  1 |   NULL |          NULL |
|  2 |      2 |          NULL |
|  3 |      5 |             4 |
+----+--------+---------------+
```

Pour obtenir des informations de typage au niveau d'un champ en particulier, vous pourrez utiliser **JSON_TYPE(doc_json)** capable de retourner : **OBJECT**, **ARRAY**, **BOOLEAN**, **NULL** ou un type SQL. La requête suivante retourne les types des différents champs des documents sélectionnés.

Fichier

```
mysql> SELECT id,
    ->        JSON_TYPE(doc) AS type_doc,
    ->        JSON_TYPE(JSON_EXTRACT(doc, '$.sequence'))
AS type_sequence,
    ->        JSON_TYPE(JSON_EXTRACT(doc, '$.desc_vol'))
AS type_desc_vol,
    ->        JSON_TYPE(JSON_EXTRACT(doc, '$.desc_vol.sieges'))
AS type_sieges,
    ->        JSON_TYPE(JSON_EXTRACT(doc, '$.desc_vol.sieges[0].
siege')) AS type_siege
    -> FROM   tab_vol_json WHERE id IN (1,4);
+----+----------+---------------+---------------+-------------+------------+
| id | type_doc | type_sequence | type_desc_vol | type_sieges | type_siege |
+----+----------+---------------+---------------+-------------+------------+
|  1 | NULL     | NULL          | NULL          | NULL        | NULL       |
|  4 | OBJECT   | INTEGER       | OBJECT        | ARRAY       | STRING     |
+----+----------+---------------+---------------+-------------+------------+
```

La fonction **JSON_VALID(expr)** permet de statuer sur la validité en terme de structure d'un document JSON. Il apparaît ainsi que deux documents sont valides et que la chaîne de caractères de la première ligne de la table n'est pas bien structurée au sens JSON.

Fichier

```
mysql> SELECT id, utilisateur,
    ->        JSON_VALID(doc) AS doc_valide,
    ->        JSON_VALID(doc_chaine) AS chaine_valide
    -> FROM   tab_vol_json WHERE id < 4
    -> ORDER BY id;
+----+-------------+------------+---------------+
| id | utilisateur | doc_valide | chaine_valide |
+----+-------------+------------+---------------+
|  1 | Brouard     |       NULL |             0 |
|  2 | Brouard     |          1 |          NULL |
|  3 | Salais      |          1 |          NULL |
+----+-------------+------------+---------------+
```

8. MISES À JOUR DANS LA TABLE

C'est toujours l'instruction **UPDATE** qu'il faudra utiliser en combinant différentes fonctions permettant de composer du contenu JSON. L'instruction suivante permet de modifier à la fois la date d'un vol et d'ajouter un passager.

Fichier

```
mysql> UPDATE tab_vol_json
    -> SET doc =
    ->    (SELECT JSON_REPLACE(doc, '$.desc_vol.sieges',
    ->               JSON_ARRAY_APPEND(
    ->                 JSON_EXTRACT(doc, '$.desc_vol.sieges'), '$',
    ->                   JSON_OBJECT('prenom','Antoine','nom','Derouin','siege',
'5A')))) ,
```

```
    ->     doc = (SELECT JSON_REPLACE(doc, '$.date_vol', '2016-07-24'))
    -> WHERE id = 3;
Query OK, 1 row affected (0.02 sec)
Rows matched: 1  Changed: 1  Warnings: 0

-- verification
mysql> SELECT id, JSON_EXTRACT(doc,'$.num_vol','$.date_vol') AS elements_vol,
    ->           JSON_EXTRACT(doc,'$.desc_vol.sieges[*].siege') AS passagers
    -> FROM tab_vol_json
    -> WHERE id = 3;
+----+--------------------------+----------------------------------------+
| id | elements_vol             | passagers                              |
+----+--------------------------+----------------------------------------+
|  3 | ["AF6143", "2016-07-24"] | ["15A", "15B", "15C", "13C", "5A", "5A"] |
+----+--------------------------+----------------------------------------+
```

Il est aussi possible de construire un document à partir d'une chaîne à l'aide de la fonction universelle de conversion **CAST**.

Fichier

```
mysql> UPDATE tab_vol_json
    ->         SET doc = CAST('{"date_vol":"2016-07-18","num_
vol":"AF6140"}' AS JSON)
    -> WHERE  id = 1;
Query OK, 1 row affected (0.00 sec)
Rows matched: 1  Changed: 1  Warnings: 0

-- verification
mysql> SELECT id,doc FROM tab_vol_json WHERE id = 1;
+----+--------------------------------------------------+
| id | doc                                              |
+----+--------------------------------------------------+
|  1 | {"num_vol": "AF6140", "date_vol": "2016-07-18"} |
+----+--------------------------------------------------+
```

9. OPÉRATEURS DE COMPARAISON

Des contenus JSON peuvent être comparés avec la plupart des opérateurs (**=**, **<**, **<=**, **>**, **>=**, **<>** et **!=**), mais pas avec les opérateurs **BETWEEN**, **IN**, **GREATEST** et **LEAST**. La comparaison s'opère en deux étapes. La première est basée sur les valeurs des types JSON, s'ils diffèrent, le résultat dépendra de leur hiérarchie. Si les deux valeurs comparées sont de même type, alors le résultat dépendra de l'échelle du type en question. La hiérarchie des types est la suivante, du plus fort au plus faible : **BLOB**, **BIT**, **OPAQUE**, **DATETIME**, **TIME**, **DATE**, **BOOLEAN**, **ARRAY**, **OBJECT**, **STRING**, **INTEGER**/**DOUBLE**, **NULL**.

La requête suivante compare des fragments de contenus JSON, le premier est un entier, le deuxième une chaîne, le troisième un objet et le quatrième un tableau (le plus fort dans la précédence des types ici représentés).

Fichier

```
mysql> SELECT id,
    ->        JSON_EXTRACT(doc, '$.sequence')            AS seq,
    ->        JSON_EXTRACT(doc,'$.num_vol')              AS num_vol,
    ->        JSON_TYPE(doc)                             AS type_doc,
    ->        JSON_EXTRACT(doc,'$.num_vol','$.date_vol') AS elements_vol
    -> FROM   tab_vol_json
    -> WHERE  id IN (3,4)
     -> AND    JSON_EXTRACT(doc,'$.num_vol')               > JSON_
EXTRACT(doc, '$.sequence')
     -> AND    JSON_EXTRACT(doc,'$.num_vol','$.date_vol') > JSON_
EXTRACT(doc,'$.num_vol')
     -> AND    JSON_EXTRACT(doc,'$.num_vol','$.date_vol') > doc;
+----+------+----------+----------+--------------------------+
| id | seq  | num_vol  | type_doc | elements_vol             |
+----+------+----------+----------+--------------------------+
|  3 | 345  | "AF6143" | OBJECT   | ["AF6143", "2016-07-24"] |
|  4 | 346  | "AF6146" | OBJECT   | ["AF6146", "2016-07-19"] |
+----+------+----------+----------+--------------------------+
```

Les clauses **ORDER BY** et **GROUP BY** répondent également à cet ordre.

Fichier

```
mysql> SELECT id, JSON_EXTRACT(doc,'$.desc_vol.porte')       AS num_porte,
    ->            JSON_EXTRACT(doc,'$.num_vol','$.date_vol') AS elements_
vol
    -> FROM   tab_vol_json
    -> WHERE  id IN (2,3,4)
     -> ORDER BY num_porte, elements_vol;
+----+-----------+--------------------------+
| id | num_porte | elements_vol             |
+----+-----------+--------------------------+
|  2 | NULL      | ["AF6141", "2016-07-18"] |
|  3 | "20A"     | ["AF6143", "2016-07-24"] |
|  4 | "20A"     | ["AF6146", "2016-07-19"] |
+----+-----------+--------------------------+
```

10. INDEXATION

Depuis la version 5.7.8, et au niveau du moteur InnoDB, l'indexation secondaire sur des colonnes virtuelles est possible. Un index secondaire peut être créé à partir d'une ou de plusieurs colonnes virtuelles ou une combinaison de colonnes virtuelles ou non. Un index sur une colonne virtuelle peut être défini de manière unique.

Le script suivant présente deux index uniques, le premier porte sur le champ **sequence** (numérique), l'autre concerne le champ **_id** (chaîne de caractères) qui nécessite de supprimer les doubles quotes pour chaque valeur.

Fichier

```
mysql> ALTER TABLE tab_vol_json
    ->       ADD doc_sequence INT
     ->        GENERATED ALWAYS AS JSON_EXTRACT(doc, '$.sequence'));
Query OK, 0 rows affected (0.47 sec)
```

```
mysql> CREATE UNIQUE INDEX index_doc_seq ON tab_vol_json(doc_sequence);
Query OK, 0 rows affected (0.20 sec)

mysql> ALTER TABLE tab_vol_json
    ->       ADD doc_id VARCHAR(8)
     ->        GENERATED ALWAYS AS (JSON_UNQUOTE(JSON_EXTRACT(doc,
'$._id')));
Query OK, 0 rows affected (0.06 sec)

mysql> CREATE UNIQUE INDEX index_doc_id ON tab_vol_json(doc_id);
Query OK, 0 rows affected (0.09 sec)
```

Une fois créées, les requêtes qui testent ces nouvelles colonnes pourront bénéficier d'une indexation efficace.

Fichier

```
mysql> EXPLAIN SELECT id,doc_id, JSON_EXTRACT(doc,'$.num_vol','$.date_vol')
AS elements_vol
    ->         FROM   tab_vol_json
     ->          WHERE  doc_id = 'FE345_90';
+----+-------------+--------------+-----+---------------+--------------+----+
| id | select_type | table        | ... | possible_keys | key
| key_len | ...                                                            |
+----+-------------+--------------+--------------------+--------------+----+
|  1 | SIMPLE      | tab_vol_json | ... | index_doc_id  | index_doc_id
| 27      | ...  |+-------------+--------------+-----+---------------+----+
```

CONCLUSION

Le terme *Document Store* introduit depuis la version 5.7.12 inclut également une nouvelle interface de commandes *MySQL Shell* et des méthodes d'accès à des collections JSON accessibles à l'aide de **JavaScript**, **Python** ou SQL. Présente depuis près de 4 ans, la version 5.7 va céder la place à la version 8 qui ne semble pas amener de grandes avancées autour des données non structurées. Citons le nouvel opérateur **->>** qui équivaut à utiliser **JSON_UNQUOTE()** sur le résultat de **JSON_EXTRACT()**. Enfin, les nouvelles fonctions **JSON_ARRAYAGG()** et **JSON_OBJECTAGG()** permettront d'agréger du contenu JSON dans un tableau ou dans un objet. ■

RÉFÉRENCES

[1] Site officiel de MySQL :
http://dev.mysql.com/doc/refman/5.7/en/

[2] Site officiel de MySQL Server Blog :
http://mysqlserverteam.com/category/json/

[3] C. Soutou, *«Programmer avec MySQL »*, Eyrolles, 5e édition à venir 2017.

VISITEZ NOTRE NOUVELLE BOUTIQUE ET DÉCOUVREZ NOS GUIDES !

LINUX MAGAZINE / FRANCE | GNU LINUX PRATIQUE ESSENTIEL | MISC Multi-System & Internet Security Cookbook | HACKABLE MAGAZINE | Open Silicium

ET VOUS ?
COMMENT LISEZ-VOUS VOS MAGAZINES PRÉFÉRÉS ?

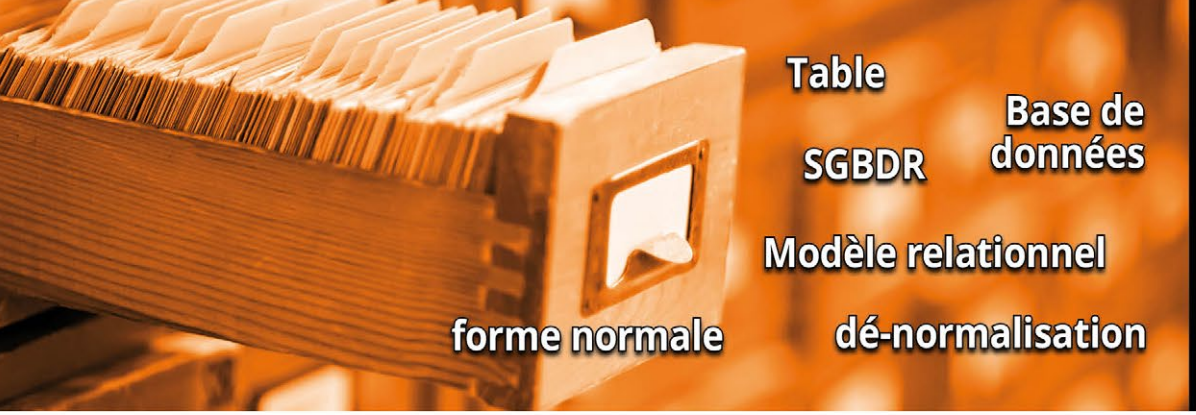
« Moi, je les lis en version PAPIER ! »

« Moi, je les lis en version PDF ! »

« Moi, je consulte la BASE DOCUMENTAIRE ! »

RENDEZ-VOUS SUR **www.ed-diamond.com**

POUR DÉCOUVRIR TOUTES LES MANIÈRES DE LIRE VOS MAGAZINES PRÉFÉRÉS !



ÉTAPE 1

Je conçois ma base de données

ÉTAPE 2

J'installe mon SGBDR MySQL MariaDB

ÉTAPE 3

J'utilise le langage SQL pour accéder à ma base

ÉTAPE 4

Je sécurise ma base et l'accès à mes données

ÉTAPE 5

Je crée des programmes utilisant mes bases de données