N°67 JUILLET - AOÛT 2013

GNU LINUX MAGAZINE / FRANCE HORS SÉRIE

L 15066 - 67 H - F: 8,00 € - RD

ADMINISTRATION ET DÉVELOPPEMENT SUR SYSTÈMES OPEN SOURCE ET EMBARQUÉS

MYSQL & BASES DE DONNÉES

TOUT CE QUE VOUS DEVEZ SAVOIR POUR INSTALLER ET EXPLOITER EFFICACEMENT VOS BASES

NIVEAU DÉBUTANT À INTERMÉDIAIRE

LA THÉORIE

LES OUTILS

LA PROGRAMMATION

France METRO : 8 € – CH : 13,80 CHF – BEL/PORT.CONT : 8,90 € – DOM : 8,50 € – CAN : 13,99 $ cad – NCAL/S : 1100 CFP – POL/A : 1600 CFP – TUNISIE : 17,40 TND – MAR : 98 MAD

DEVENEZ UN VRAI SORCIER DE LA LIGNE DE COMMANDES !

DISPONIBLE DÈS FIN JUILLET

25 formules incluses

Faire ses premiers pas dans le terminal

Comprendre la syntaxe d'une commande

Découvrir les options du terminal

Comment naviguer parmi ses fichiers ?

Comment écrire et exécuter un script shell ?

Comment créer et supprimer des répertoires ?

Comment lire le contenu d'un fichier ?

Sous réserve de toutes modifications.

LINUX PRATIQUE HORS-SÉRIE N°27

DISPONIBLE DÈS FIN JUILLET CHEZ VOTRE MARCHAND DE JOURNAUX ET SUR : ed-diamond.com

SOMMAIRE
N°67

LA THÉORIE

LES OUTILS

SGBD & LANGAGES

ABONNEMENTS & COMMANDES

ÉDITO

Nous recevons tous, chaque jour, de très nombreux mails d'entreprises qui, étrangement, paraissent bien nous connaître... Votre mémoire est défaillante ? Voici quelques extraits de mails qui devraient vous rafraîchir la mémoire :

« D'après vos derniers achats, le site AchètePlusDeLivres.com vous recommande :
- Polar : « Bilou m'a tuer » de Lin Hioux.
- Vie pratique : « On m'a donné un Mac, comment m'en débarrasser ? » de Petro Gnon della Pommas.
- Cuisine : « Comment cuisiner le pingouin ? 20 recettes pour petits et grands » de Cyril Inyak. »

« Pendant votre absence des hordes de trolls ont attaqué vos alliés !
Deux forts ont été perdus et cinq troupes ont été anéanties.
Connectez-vous pour combattre l'ennemi ! »

« Depuis que vous n'êtes plus venu sur Livreportait.com vous avez manqué :
- 75 demandes d'amis
- 1232 nouveaux messages dont voici des extraits :
- Je vais à la boulangerie.
- Il y a du monde, j'attends...
- J'attends encore.
- J'achète du pain.
- Je suis sortie de la boulangerie.
Imaginez tout ce que vous risquez encore de rater... reconnectez-vous vite ! »

Le point commun de tous ces mails ? Les bases de données bien sûr ! Notre vie est en permanence enregistrée et analysée par des serveurs. Le stockage de ces informations est réalisé dans des bases de données plus ou moins sécurisées, à partir desquelles de nouvelles connaissances sont extraites. On apprend régulièrement que des listes de comptes utilisateurs ont été piratées sur tel ou tel serveur, mettant à la disposition de hackers bon nombre d'informations personnelles. Très récemment, une petite controverse a vu le jour : la NSA (agence nationale de sécurité américaine) et le FBI (bureau fédéral d'enquête, lui aussi américain) auraient un accès direct aux serveurs de Google, Facebook, etc. Que l'information soit vérifiée ou pas n'a pas une grande importance : à partir du moment où des données privées sont hébergées en clair sur des serveurs appartenant à des sociétés qui peuvent marchander ces-dites données, les informations qu'elles contiennent sont susceptibles d'être utilisées à notre insu. Nous perdons tout contrôle sur une information proprement stockée et archivée dans des bases de données et que nous ne pouvons même pas effacer avec certitude !

En France, nous avons un organisme qui est chargé de veiller à ce que l'informatique soit au service du citoyen et qu'elle ne porte atteinte ni à l'identité humaine, ni aux droits de l'homme, ni à la vie privée, ni aux libertés individuelles ou publiques. Il s'agit de la CNIL, Commission Nationale de l'Informatique et des Libertés. Les autres pays européens disposent également de CNIL. Dernièrement, six de ces CNIL, dont la CNIL française, ont engagé une action contre Google pour ne pas avoir revu ses règles de confidentialité. Mais alors que le Web est mondial et que les sociétés sont bien souvent implantées à l'étranger, quel est le véritable poids des CNIL ?

Lorsque l'on doit manipuler des données, il est donc essentiel de bien comprendre le fonctionnement des bases de données. C'est ce que nous vous proposons dans ce hors-série.

Une dernière précision avant de vous laisser profiter de ce magazine : il existe de nombreux systèmes de gestion de bases de données et dans ce hors-série, nous avons choisi de n'en privilégier aucun. Les petites guéguerres entre pro-PostgreSQL, pro-MySQL, ou encore pro-NoSQL ne sont pas au centre de ce numéro. Nous nous attacherons uniquement à une compréhension théorique et pratique des concepts fondamentaux. À un moment donné, vous devrez faire un choix en fonction du projet et non en fonction d'un attachement fanatique à tel ou tel système...

Tristan Colombo

GNU/Linux Magazine France Hors-série
est édité par Les Éditions Diamond

LES ÉDITIONS DIAMOND
B.P. 20142 – 67603 Sélestat Cedex
Tél. : 03 67 10 00 20 – **Fax :** 03 67 10 00 21
E-mail : lecteurs@gnulinuxmag.com
Service commercial : abo@gnulinuxmag.com
Sites : www.gnulinuxmag.com – www.ed-diamond.com
Directeur de publication : Arnaud Metzler
Rédacteur en chef : Denis Bodor
Secrétaire de rédaction : Véronique Sittler
Réalisation graphique : Kathrin Scali

Responsable publicité : Valérie Fréchard, Tél. : 03 67 10 00 27
v.frechard@ed-diamond.com
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 : À parution, **N° ISSN :** 0183-0864
Commission paritaire : K78 976
Périodicité : Bimestrielle
Prix de vente : 8,00 €

QU'EST-CE QU'UNE BASE DE DONNÉES ?

par Tristan Colombo

De nombreuses applications que nous utilisons quotidiennement stockent des informations dans des bases de données. Nous savons donc intuitivement à quoi elles servent... Mais avant d'en utiliser une, il serait bon de savoir exactement ce que signifie le terme « base de données ».

1 Qu'est-ce qu'une base de données ?

Une base de données sert à stocker sur un support permanent des données, c'est-à-dire des informations. Mais il ne s'agit pas simplement d'un ou de plusieurs fichier(s) contenant des données brutes ! Dans une base de données, il y a en plus une notion fondamentale : les données sont **structurées**.

Prenons un exemple : nous souhaitons stocker une liste de clubs de volley-ball. Pour chacun de nos clubs, nous allons conserver le nom, la ville et, bien sûr, la division. La structure définissant ce qu'est un club est donc la suivante : nom (chaîne de caractères), ville (chaîne de caractères) et division (entier). Par la suite, nous nous apercevons que nous souhaitons aussi connaître la liste des joueurs évoluant dans ces différents clubs. Nous allons alors définir une nouvelle structure permettant de stocker les informations relatives à un joueur : nom, prénom, date de naissance, club, etc. Il existe une **association** entre un club et un joueur puisqu'un club « possède » un joueur et qu'un joueur « appartient » à un club (voir figure 1). Les clubs et les joueurs sont des **entités** de la base de données.

Fig. 1 : Représentation des structures de données relatives aux clubs et joueurs de volley-ball

Pour dialoguer avec la base de données, plutôt que d'avoir à écrire des programmes complexes, pour pouvoir ajouter, modifier, rechercher des données, on utilise des programmes regroupés dans un **système de gestion de base de données** (abrégé en SGBD). En plus des actions précédentes, le SGBD est chargé de garantir la **sécurité** et l'**intégrité** des données et de permettre un accès **centralisé** aux données. En effet, si tous les développeurs pouvaient manipuler directement les fichiers de la base, il serait impossible de garantir l'intégrité des données. De même, si plusieurs utilisateurs demandent à accéder aux mêmes fichiers au même moment, des problèmes de concurrence d'accès peuvent se poser. Par exemple, il faut garantir l'intégrité des données (tout en évitant le blocage du système) si deux utilisateurs tentent de modifier la même donnée en même temps.

Pour interroger la base de données à travers le SGBD, l'utilisateur (développeur, administrateur de la base de données) va poser une question, exprimer une **requête**. Pour instaurer ce dialogue homme/base de données, il faut un langage simple qui ne soit pas trop compliqué à apprendre car ne s'adressant pas forcément qu'à de purs informaticiens. Ce langage, le **langage de requêtes**, est lié au SGBD. Le plus connu des langages de requêtes est le SQL.

Pour résumer, le SGBD peut être vu comme un ensemble de programmes disposés en trois couches (voir figure 2) :

- la couche la plus basse est constituée par le gestionnaire de fichiers : c'est la **couche physique**. Cette couche fournit aux couches supérieures des accès aux données stockées dans les fichiers et permet d'effectuer des recherches efficaces dans le contenu des objets grâce à des mécanismes d'indexation.
- la couche intermédiaire, appelée **couche interne**, est dédiée aux manipulations logiques : gestion des données stockées dans la couche inférieure (ajout, modification, etc.) et liens entre les objets et leur structure, de manière à optimiser les accès auxdits objets. Le maintien de l'intégrité et de la sécurité, ainsi que le langage de requête se trouvent dans cette couche.
- la dernière couche est la **couche externe**. Sa fonction essentielle est de fournir des outils de haut niveau, de mettre en forme les données avant de les renvoyer vers les programmes utilisant la base.

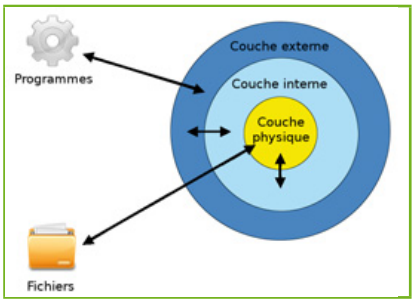


Fig. 2 : Les trois couches du système de gestion de base de données

Dans la base de données, on distingue la structure des informations qui y seront stockées (on parle de **modèle de données** ou de **schéma de données**) des informations elles-mêmes, assurant ainsi une indépendance logique et physique des programmes par rapport aux données.

Lorsque l'on modélise des données pour pouvoir les insérer dans une base, on commence par les représenter telles qu'elles sont réellement avant de modéliser la façon dont elles seront stockées en machine. On distingue alors le **modèle conceptuel de données** (ou MCD) du **modèle logique de données** (ou MLD) et du **modèle physique de données** (ou MPD). Nous reviendrons sur ces notions dans l'article traitant de la modélisation avec la méthode MERISE. Pour l'instant, sachez simplement qu'il y a le modèle conceptuel de données qui établit les relations entre objets sans se soucier du SGBD qui sera utilisé, alors que le modèle logique de données et le modèle physique de données tiennent compte du SGDB.

Définition

Une **base de données** est un ensemble structuré permettant de stocker et de manipuler simplement une grande quantité d'informations. Par « manipuler », on entend insérer, supprimer, modifier et rechercher des données. Les informations stockées dans la base modélisent des objets dont la description est distincte des valeurs les définissant.

Si nous reprenons l'exemple précédent des clubs et joueurs de volley-ball, le MCD pourra être défini à partir des informations suivantes :

```
Entités :
  Club(nom, ville, division)
  Joueur(nom, prénom, date_de_naissance)
Association :
  Appartenir(Joueur, Club, fin_de_contrat)
```

Définition

Un **système de gestion de base de données** (SGBD) est un logiciel permettant de manipuler les informations dans la base de données en garantissant une indépendance physique et logique des programmes envers les données, l'intégrité et le partage des données.

2 Historique

En 1956 l'outil indispensable à l'apparition des bases de données est lancé par IBM : il s'agit du disque dur et à cette époque, il a une capacité de 5Mo (soit à peu près deux fichiers MP3 pas trop longs).

Jusqu'aux années 1960, les données étaient structurées dans des fichiers éventuellement reliés entre eux. Mais à cette même époque, on assiste à l'apparition d'applications de plus en plus lourdes devant gérer de plus en plus de données... D'où la nécessité de séparer la structure des données des outils de manipulation. En 1965, Charles Bachman définit l'architecture Ansi/Sparc sur laquelle repose encore les SGBD modernes : les trois couches vues précédemment. On peut donc dire que c'est à cette date que naît la première génération de SGBD. Pour la petite histoire, Charles Bachman a reçu le prix Turing pour ses contributions sur les technologies des bases de données.

En 1970 naît la deuxième génération de SGBD basée sur les travaux de Edgar F. Codd : le modèle relationnel (nous aborderons ce sujet en détail dans l'article suivant). C'est avec cette génération qu'apparaît le langage de requêtes SQL. Rendue publique à partir des années 1980, cette famille de SGBD est celle qui est encore à l'heure actuelle la plus utilisée.

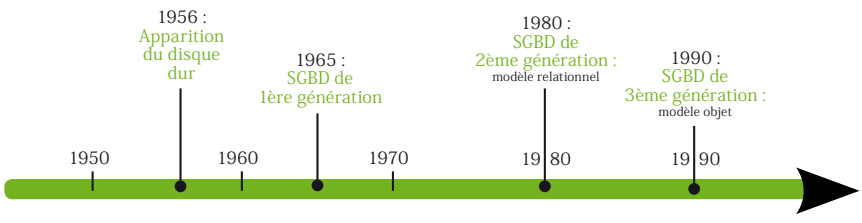
Dans les années 1990, c'est au tour de la troisième génération de SGBD d'apparaître : les bases de données orientées objet permettant de manipuler plus simplement les données (objets) dans des programmes écrits en langages orientés objet. Les SGBDO de troisième génération ne sont que très faiblement utilisés par rapport aux SGBD de deuxième génération et ne remplaceront vraisemblablement jamais ces derniers.

Nous attendons maintenant la quatrième génération... Peut-être sera-t-elle basée sur les accès et le traitement des données du Web ? ■

Définition

Base de données et banque de données, quelle différence ?

Une banque de données est un ensemble de données relatif à un domaine particulier. Ces données peuvent être conservées sur différents supports qui ne sont pas forcément numériques. Dans le cas d'un support numérique, elles peuvent être éventuellement organisées en base de données.



LES BASES DE DONNÉES RELATIONNELLES

par Tristan Colombo

Le modèle relationnel est le plus connu et le plus utilisé. Il a été décrit en 1970, en pleine explosion des besoins de stockage d'informations, par Edgar Codd, pour séparer la représentation logique des données de leur organisation physique.

1 Objectifs du modèle relationnel

Les objectifs formulés par Edgar Codd et atteints dans le modèle relationnel sont les suivants :

- Assurer l'indépendance des données : des modifications du schéma physique (manière dont les données sont stockées sur le disque) n'interfèrent pas avec le schéma logique ou conceptuel (organisation des données dans la base), ni même avec la vue que les utilisateurs ont des données.
- Gérer les problèmes de cohérence et de redondance des données.

Le modèle relationnel est basé sur des règles édictées par Edgar Codd et appelées « les douze règles de Codd » **[1] [2]**. Si une seule de ces règles n'est pas respectée, le SGBD ne peut pas être considéré comme relationnel. Ce sont un peu les dix commandements des systèmes de gestion de bases de données relationnelles, à la différence près que dans les dix commandements, il y a bien dix commandements... Les douze règles de Codd sont numérotées de zéro jusqu'à douze, ce qui fait donc treize règles (superstition anglo-saxonne ?). Plutôt que de lister ces règles simplement traduites de l'anglais, je vous en propose une interprétation.

Règle 0 : Introduction

Un système de gestion de bases de données relationnel (SGBDR) ne peut effectuer que des opérations relationnelles pour chacune de ses tâches. L'utilisation de caractéristiques techniques pour définir des éléments de la base de données représente par exemple une violation de cette règle.

Règle 1 : Unicité

Toutes les données d'une base de données relationnelle sont représentées de la même manière par des valeurs dans des tables : les colonnes correspondant à un type de donnée et les lignes sont les différents enregistrements dans la table. Par exemple, la table `Livres` contiendra les colonnes `Titre`, `Auteur`, `ISBN`, etc. `(A Relational Model of Data for Large Shared Data Banks, Edgar Codd, 0-201-14192-2, ...)` pourra être un enregistrement de cette table.

Règle 2 : Garantie d'accès

À partir de trois informations de base, le SGBDR doit être capable de retrouver n'importe quelle donnée. Ces informations sont le nom de la table, le nom d'une colonne et la valeur d'une clé dite **primaire**. Une clé primaire est une colonne dont la valeur permet d'identifier de manière unique un enregistrement. Sa valeur doit donc forcément être unique pour chaque enregistrement. En reprenant l'exemple de la table `Livres`, on pourrait dire que la colonne `ISBN` est la clé primaire de la table : deux livres n'auront jamais le même identifiant, alors qu'un auteur peut avoir publié plusieurs livres ou qu'un même titre peut avoir été utilisé par plusieurs auteurs.

Règle 3 : Les valeurs nulles

Lorsqu'une information relative à un champ d'une table (valeur d'un enregistrement associée à une colonne) est manquante, on utilisera le marqueur `NULL`. Ce marqueur permet de rendre optionnelle la présence d'une information dans un champ et de ne pas venir interférer avec d'autres données. En effet, supposons que la valeur 0 soit interprétée comme une absence de donnée : que se passe-t-il si le champ représente une note et que nous souhaitons calculer une moyenne ? S'agit-il d'une absence à un examen ou d'un examen complètement raté ? Le marqueur `NULL` permet de distinguer ces deux cas.

Règle 4 : Catalogue relationnel

La description de la base et des données est réalisée de la même manière. Ainsi, on utilisera un seul et même langage d'interrogation pour accéder aux informations sur la structure de la base ou sur les données elles-mêmes.

Règle 5 : Langage de requête complet

Un SGBDR doit posséder au moins un langage de requête qui permette :

- de manipuler le schéma de la base ;
- de manipuler les données ;

- de garantir l'intégrité des données et de respecter les règles d'accès ;
- de gérer les transactions (débuter, valider ou annuler une transaction).

Le langage SQL, que nous aborderons dans un article dédié, respecte complètement ces règles.

Règle 6 : Mise à jour des vues

Une vue est une sorte de table virtuelle créée à partir d'une requête. Ce mécanisme permet de ne pas retaper systématiquement une même requête et de présenter les données comme s'il s'agissait de tables. En théorie, on ne devrait accéder aux données qu'en utilisant des vues. En pratique, on manipule bien souvent directement les tables.

La règle 6 indique que le SGBDR doit être capable d'insérer, modifier ou supprimer des enregistrements d'une vue pouvant théoriquement être mise à jour (doit respecter certaines contraintes lors de sa création).

Règle 7 : Insertion, modification et suppression par lots

Le SGBDR doit être capable d'effectuer des opérations d'insertion, suppression et modification par lots. Les données doivent pouvoir être manipulées sous la forme d'ensemble d'enregistrements ou tuples, issus d'une ou plusieurs table(s).

Règle 8 : Indépendance physique des données

La modification de structure physique des données (déplacement de fichiers, renommage, etc.) ne doit avoir aucune influence sur des applications existantes. Si dans un programme nous demandons la liste des ouvrages présents dans la table **Livres**, que ceux-ci se trouvent dans le fichier **x8212** ou **zk22**, nous devons obtenir le même résultat sans indiquer dans quel fichier rechercher les informations.

Règle 9 : Indépendance logique des données

La modification de la structure de tables ne doit avoir aucun impact sur des applications qui n'utilisent que des éléments qui ne sont pas modifiés. Par exemple, l'ajout d'une colonne à une table n'aura aucun impact, puisque la donnée ne sera forcément pas utilisée dans un programme existant.

Règle 10 : Indépendance d'intégrité

Les contraintes d'intégrité d'une base de données relationnelle sont définies et stockées dans la base. Ces contraintes sont essentielles, car ce sont elles qui sont chargées d'éviter que la base ne soit corrompue par l'ajout de doublons, la disparition de valeurs, etc., qui surviennent au cours de son utilisation. Par exemple, si deux enregistrements sont liés, la suppression de l'un doit entraîner la suppression de l'autre. Si nous supprimons un auteur de notre base, est-il cohérent de conserver la liste de ses ouvrages ?

Règle 11 : Indépendance de distribution

La répartition de diverses parties d'une base de données doit rester totalement transparente pour les programmes qui l'utilisent.

Règle 12 : Non subversion

Un langage de plus bas niveau que le langage de requêtes ne doit pas permettre de contourner les règles de sécurité ou d'intégrité.

2 L'algèbre relationnelle

L'algèbre relationnelle est une théorie mathématique elle-même basée sur la théorie des ensembles. Cette théorie se trouve à la base du modèle relationnel. À partir de trois notions de base, on définit cinq opérateurs permettant de manipuler les ensembles de données, les autres opérateurs pouvant être obtenus par composition des cinq premiers.

2.1 Notions de base

Avant de pouvoir stocker des valeurs dans des champs, il faut savoir quel type auront ces valeurs. Pour cela, on va définir des **domaines**, qui sont des ensembles de valeurs associés à un nom. Par exemple, **INTEGER** est le domaine contenant des entiers, **BOOLEAN** est un ensemble contenant vrai ou faux, etc. Ces domaines sont les domaines de base, mais il est possibwle de définir de nouveaux domaines puisqu'il ne s'agit que d'ensembles de valeurs. On pourrait par exemple créer le domaine **SPORT** contenant **{ TENNIS, VOLLEYBALL, CYCLISME, ATHLETISME }**.

À partir de la notion de domaine, nous allons pouvoir définir la notion fondamentale du modèle relationnel : la **relation**. Une relation est une table où chaque colonne possède un nom et un domaine associé. Plus formellement, on parlera de sous-ensemble du produit cartésien d'une liste de domaines caractérisé par un nom. La représentation la plus simple étant une table à deux dimensions, on parlera souvent de « table » et non de « relation », comme nous l'avons fait en première approximation dans la section précédente.

Prenons pour exemple la relation **Sportif** permettant de stocker des informations sur des sportifs :

Nom	Sport
Federer	TENNIS
Mastrangello	VOLLEYBALL

La colonne **Nom** prend ses valeurs dans le domaine « chaînes de caractères » et la colonne **Sport** prend ses valeurs dans le domaine **SPORT**.

Un **tuple** correspond à la ligne d'un enregistrement dans une table. On peut également trouver le terme « vecteur » pour qualifier cette information. Dans la relation **Sportif**, **(Federer, TENNIS)** est un tuple.

Enfin, la dernière notion que nous aborderons dans cette partie est la notion de **clé**. L'algèbre relationnelle se basant sur la théorie des ensembles, comme un ensemble ne peut contenir deux fois le même élément, une relation ne peut contenir deux fois le même tuple. Pour assurer cette unicité, on utilise la notion de clé : attribut ou colonne permettant d'identifier de manière unique un tuple.

S'il existe plusieurs clés, toutes peuvent être utilisées indépendamment pour cibler un tuple. On en choisit alors une arbitrairement, que l'on nomme alors **clé primaire**. Par abus de langage, même lorsqu'il n'existe qu'une clé on parle également de clé primaire.

Dans une relation, un attribut d'un tuple peut faire référence à un tuple d'une relation extérieur. Chacun des deux tuples possède une clé primaire, mais il faut pouvoir faire référence au tuple « extérieur » de manière unique et on utilise donc encore une clé, qui cette fois-ci est qualifiée de **clé étrangère** (faisant référence à une relation étrangère à la relation courante).

Prenons un nouvel exemple : nous souhaitons stocker des références cinématographiques et y associer des réalisateurs. Nous aurons deux relations :

- **Film(idFilm, Titre, Année, idReal)**
- **Realisateur(idRealisateur, Nom, Prenom, Annee_Naissance)**

Ici, **idFilm** est la clé primaire de **Film**, **idRealisateur** est la clé primaire de **Realisateur** et, dans la relation **Film**, **idReal** est la clé étrangère pointant sur la relation **Realisateur** et faisant référence à **idRealisateur**. En peuplant nos tables, nous y verrons plus clair. Nous avons tout d'abord la table **Film** :

idFilm	Titre	Annee	idReal
1	Iron Man 3	2013	2
2	Le Hobbit : un voyage inattendu	2012	1

Puis la table **Realisateur** :

idRealisateur	Nom	Prenom	Annee_ Naissance
1	Jackson	Peter	1961
2	Black	Shane	1961

On peut voir que « Iron Man 3 » a été tourné par le réalisateur dont la clé primaire est 2. Il s'agit donc de Shane Black.

2.2 Opérateurs

Les opérations de base de l'algèbre relationnelle sont définies à partir de cinq opérateurs. Parmi ces opérateurs, on trouve trois opérateurs binaires (l'union, la différence et le produit cartésien) et deux opérateurs unaires (la sélection et la projection).

2.2.1 Union

L'opération d'union de deux relations est la même qu'en théorie des ensembles : on récupère l'ensemble des tuples appartenant à l'une, à l'autre ou aux deux relations. Pour que cet opérateur puisse être appliqué, il faut que les deux relations aient exactement le même schéma (même domaine de définition des tuples et tuples de même taille). Prenons pour exemple deux relations **Sportif1** et **Sportif2** :

- **Sportif1**

Nom	Sport
Federer	TENNIS
Mastrangello	VOLLEYBALL

- **Sportif2**

Nom	Sport
Nadal	TENNIS
Djokovic	TENNIS

Nous pouvons obtenir l'union de ces deux relations par **Sportif1 U Sportif2** (pouvant être également notée **UNION(Sportif1, Sportif2)**) :

Nom	Sport
Federer	TENNIS
Mastrangello	VOLLEYBALL
Nadal	TENNIS
Djokovic	TENNIS

2.2.2 Différence

La différence de deux relations est l'ensemble des tuples de la première relation qui sont absents dans la seconde. Attention : cet opérateur est **non commutatif**, les relations doivent avoir le même schéma et produisent une relation de même schéma. Reprenons nos deux tables de sportifs légèrement modifiées :

- **Sportif1**

Nom	Sport
Federer	TENNIS
Mastrangello	VOLLEYBALL
Giba	VOLLEYBALL

Sportif2 :

Nom	Sport
Federer	TENNIS
Djokovic	TENNIS

Nous pouvons alors déterminer **Sportif1 - Sportif2** (ou **DIFFERENCE (Sportif1, Sportif2)**) :

Nom	Sport
Mastrangello	VOLLEYBALL
Giba	VOLLEYBALL

Comme cet opérateur est non commutatif, le résultat de **Sportif2 - Sportif1** sera bien sûr différent :

Nom	Sport
Djokovic	TENNIS

2.2.3 Produit cartésien

Le produit cartésien porte sur deux relations et produit une relation dont les tuples représentent toutes les combinaisons possibles des tuples des relations utilisées dans l'opération. Prenons pour exemple une table **Magazine** et une table **Version** :

- **Magazine**

Nom	Éditeur
GNU Linux Magazine	Les éditions Diamond
GNU Linux Pratique	Les éditions Diamond
Linux Pratique Essentiel	Les éditions Diamond

- **Version**

Type
Classique
Hors-série

Le résultat du produit cartésien de ces deux relations, soit **Magazine X Version** (ou encore **PRODUCT(Magazine, Version)**), est le suivant :

Nom	Éditeur	Type
GNU Linux Magazine	Les éditions Diamond	Classique
GNU Linux Magazine	Les éditions Diamond	Hors-série
GNU Linux Pratique	Les éditions Diamond	Classique
GNU Linux Pratique	Les éditions Diamond	Hors-série
Linux Pratique Essentiel	Les éditions Diamond	Classique
Linux Pratique Essentiel	Les éditions Diamond	Hors-série

Si les deux relations sur lesquelles est appliqué le produit cartésien possèdent des noms de colonne en commun, il peut y avoir une ambiguïté sur les colonnes de la relation résultante. On procède alors à un renommage des colonnes.

2.2.4 Sélection

La sélection s'applique à une seule relation et extrait les tuples qui satisfont un critère de comparaison utilisant les opérateurs **=**, **!=**, **<**, **>**, **<=** et **>=**. Prenons encore une fois la table **Sportif** :

- **Sportif**

Nom	Sport
Federer	TENNIS
Mastrangello	VOLLEYBALL
Giba	VOLLEYBALL

Si nous voulons obtenir la liste des sportifs pratiquant le volley-ball, nous calculerons **σSport=VOLLEYBALLSportif** et nous obtiendrons :

Nom	Sport
Mastrangello	VOLLEYBALL
Giba	VOLLEYBALL

2.2.5 Projection

La projection ne s'applique que sur une relation sur laquelle elle va supprimer des attributs (des colonnes). On passera alors de tuples de dimension n à des tuples de dimension p avec n > p. Prenons pour exemple une table énumérant des films :

- **Film**

idFilm	Titre	Année	Réalisateur
1	Iron Man 3	2013	Jackson
2	Le Hobbit : un voyage inattendu	2012	Black

La projection de la relation **Film** sur les attributs **idFilm** et **Titre**, notée **∏idFilm, TitreFilm**, donne :

idFilm	Titre
1	Iron Man 3
2	Le Hobbit : un voyage inattendu

2.2.6 Jointure

Même si la jointure est obtenue par composition, son rôle fondamental dans l'algèbre relationnelle fait que l'on ne pouvait pas ne pas la mentionner ici. Il s'agit d'une extension du produit cartésien sur lequel on applique une condition de sélection. Si aucune condition n'est spécifiée, il s'agira d'une jointure dite « naturelle », qui effectue le produit cartésien des deux relations pour chaque tuple ayant les mêmes valeurs pour des attributs de même nom. Prenons l'exemple des tables **Film** et **Realisateur** suivantes :

- **Film**

Titre	Année
Iron Man 3	2013
Le Hobbit : un voyage inattendu	2012

- **Realisateur**

Prénom	Nom	Titre	Tournage
Shane	Black	Iron Man 3	2012
Peter	Jackson	Le Hobbit : un voyage inattendu	2012

Une jointure naturelle de ces deux relations, notée **JOIN(Film, Realisateur)**, donne :

Titre	Année	Prénom	Nom	Tournage
Iron Man 3	2013	Shane	Black	2012
Le Hobbit : un voyage inattendu	2012	Peter	Jackson	2012

C'est la colonne **Titre** qui a servi à joindre les données des deux tables.

Nous pourrions également effectuer une jointure en précisant une condition : **JOIN(Film, Realisateur, Année = Tournage)**. Nous obtiendrions alors :

Titre	Année	Prénom	Nom	Tournage
Le Hobbit : un voyage inattendu	2012	Peter	Jackson	2012

Conclusion

Le modèle relationnel est donc basé sur les douze règles de Codd et l'algèbre relationnelle, dont les principaux opérateurs ont été présentés précédemment. Un SGBD est dit « relationnel » (SGBDR) lorsque les données sont présentées en suivant le modèle relationnel. Les requêtes exécutées par ces SGBD utilisent l'algèbre relationnelle. ■

Bibliographie

[1] CODD (Edgar), « Is your DBMS really relational ? », ComputerWorld, 1985.

[2] CODD (Edgar), « Does your DBMS run by the rules », ComputerWorld, 1985.

D'AUTRES MODÈLES DE BASES DE DONNÉES

par Tristan Colombo

Le modèle relationnel n'est pas le seul modèle existant. Nous avons mentionné précédemment le modèle objet comme faisant partie de la troisième génération de systèmes de gestion de bases de données, mais il en existe bien d'autres. Présentation succincte de quatre modèles « exotiques » : le modèle objet, le modèle déductif, le modèle hiérarchique et le modèle réseau.

1 Bases de données orientées objet

Les langages de programmation orientés objet utilisent le concept d'objet pour modéliser des éléments de leur environnement. Les bases de données orientées objet se basent sur le même formalisme et stockent de manière persistante des objets. Les objets sont des structures de données composées d'attributs et de méthodes. Les attributs se rapprochent de la notion de tables du modèle relationnel, puisqu'il s'agit des champs permettant de conserver des données dans un enregistrement. En ce qui concerne les méthodes, ce sont des fonctions qui permettent de manipuler les valeurs stockées dans les attributs. Suivant les systèmes de gestion de bases de données objet (SGBDO), les méthodes peuvent être appliquées sur les objets directement dans le mécanisme de gestion de la base.

Du fait de la persistance des objets, un attribut est ajouté à chaque objet : l'identifiant d'objet. Cet attribut, invariant, permet d'identifier de manière unique un objet tout au long de sa vie. On peut ainsi manipuler deux objets ayant les mêmes valeurs pour chacun de leurs attributs.

Les notions de visibilité, composition, d'héritage, polymorphisme et surcharge sont bien sûr mises en œuvre. L'interrogation de la base se fait à l'aide d'un langage spécifique l'OQL (*Object Query Language*) dérivé du SQL. Malheureusement, à cause de la complexité de la définition du langage, aucun des rares éditeurs de SGBDO ne l'a implémenté entièrement (ce qui peut se traduire également par des problèmes de compatibilité entre SGBDO).

Il faut noter que le passage d'un modèle relationnel à un modèle objet est assez simple, alors que la réciproque est bien plus complexe. En fait, il n'existe pas un modèle objet, mais deux :

- si l'on part des langages orientés objet et qu'on y intègre les notions des SGBD, on obtient des systèmes de gestion de bases de données orientés objet (SGBDOO) ;
- si l'on part des SGBD relationnels et que l'on y insère des notions objets, alors on obtient cette fois des systèmes de gestion de bases de données relationnels objet (SGBDRO).

Donc, les SGBDO regroupent à la fois les SGBDOO et les SGBDRO. D'un point de vue de la modélisation objet, les SGBDOO sont plus proches des concepts de base. D'un point de vue pratique, les SGBDRO sont basés sur des SGBDR qui sont éprouvés et très répandus. Dans le petit monde des SGBDO, on a donc plus de chance de rencontrer un SGBDRO qu'un SGBDOO.

2 Bases de données déductives

Un système de gestion de bases de données déductif (SGBDD) peut, comme son nom l'indique, faire des déductions basées sur des règles et des faits stockés dans la base. Les déductions faites sont elles-mêmes des règles ou des faits. Le langage de requête d'un SGBDD se base sur le calcul des prédicats et la logique du premier ordre. Il s'agit d'un langage déclaratif de type Datalog ou Prolog. Voici un exemple de règles et de faits en Prolog. Commençons par les règles :

```
1. parent(X,Y) :- pere(X, Y), homme(X).
2. parent(X, Y) :- mere(X, Y), femme(X).
3. frere(X, Y) :- parent(Z, X), parent(Z, Y), homme(X).
4. soeur(X, Y) :- parent(Z, X), parent(Z, Y), femme(X).
```

Ces règles peuvent se lire :

1. X est un parent de Y si et seulement si X est le père de Y, et que X est un homme.
2. X est un parent de Y si et seulement si X est la mère de Y, et que X est une femme.
3. X est le frère de Y si et seulement s'il existe un parent commun Z à X et Y, et que X est un homme.
4. X est la sœur de Y si et seulement s'il existe un parent commun Z à X et Y, et que X est une femme.

Voici maintenant quelques faits constituant la base de connaissances :

```
homme(Jack Geller).
homme(Ross Geller).

femme(Judy Geller).
femme(Monica Geller-Bing).

pere(Jack Geller, Ross Geller).
pere(Jack Geller, Monica Geller-Bing).
mere(Judy Geller, Ross Geller).
mere(Judy Geller, Monica Geller-Bing).
```

À partir de ces données, nous pouvons inférer de nouvelles connaissances :

```
frere(Ross Geller, Monica Geller-Bing).
soeur(Monica Geller-Bing, Ross Geller).
```

Les SGBDD peuvent donc être vus comme des systèmes experts vus du côté base de données. Il s'agit d'une extension des SGBDR.

3 Les autres modèles

Les autres modèles ont surtout un intérêt historique : d'un point de vue évolutif, ils ont été supplantés par le modèle relationnel.

3.1 Le modèle hiérarchique

Le modèle hiérarchique est basé sur une structure arborescente, où les informations sont stockées dans des nœuds qui ne possèdent qu'un seul père (voir figure 1). Chaque nœud représente une collection d'objets (le nœud lui-même et ses fils). La représentation hiérarchique induit une notion d'ordre, de hiérarchie entre les données, qui n'est pas forcément le reflet de la réalité.

Par exemple, pour obtenir un diplôme D1, il faut valider un module M. Dans un modèle hiérarchique, le module M ne pourra pas servir à valider un diplôme D2, or dans la réalité, on peut très bien avoir un module « Techniques de communication » partagé entre un diplôme L1 Math et L1 Informatique. De nos jours, les modèles hiérarchiques peuvent être utilisés pour la création d'ontologies particulières (voir encadré).

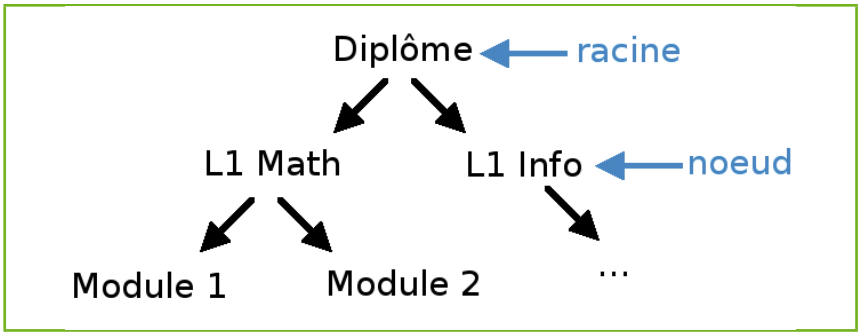


Fig. 1 : Modèle hiérarchique : chaque nœud ne possède qu'un seul père.

Ontologie

Une ontologie permet de modéliser un ensemble de connaissances sur un domaine particulier. En quoi se distingue-t-elle alors d'une simple base de données ? L'ontologie va plus loin, en s'interrogeant sur l'ensemble des termes et des relations entre les éléments d'un domaine, mais également sur les règles permettant de combiner lesdits termes et relations.

3.2 Le modèle réseau

Le modèle réseau peut être vu comme un cas particulier de modèle hiérarchique (suivant le point de vue, la réciproque est également vraie). Les données sont représentées par des enregistrements associés par des relations. La restriction de cardinalité unaire avec le père du modèle hiérarchique est supprimée. Ici, les nœuds peuvent avoir de multiples parents et enfants et les cycles sont autorisés.

Conclusion

Ces différents modèles sont très peu utilisés pour différentes raisons : ils sont soit « dépassés », dans la mesure où des solutions plus efficaces et/ou plus simples ont été trouvées, soit les éditeurs ne proposent pas de système de gestion de bases de données implémentant ces modèles. Si nous prenons le cas des systèmes de gestion de bases de données objet, les éditeurs de SGBD relationnels ont préféré intégrer les principaux avantages du modèle objet dans leur SGBD existant (et en faire un SGBD relationnel objet), plutôt que de se lancer dans l'écriture d'un SGBD orienté objet bien plus séduisant d'un point de vue théorique.

Enfin, il ne faut pas négliger non plus le facteur coût, qu'il soit en temps ou en argent, pour migrer d'un SGBD à un autre. Le modèle relationnel fonctionne plutôt bien et répond aux besoins actuels les plus courants. Les modèles objet (dans sa version orienté objet) et déductifs ne sont utilisés que dans de très rares cas. ■

LE DATA MINING OU LA FOUILLE DE DONNÉES

par Tristan Colombo

À l'époque du Far West et de la ruée vers l'or, les chercheurs d'or se précipitaient sur les filons aurifères. Il y avait ceux qui ne trouvaient rien, étaient ruinés et ceux qui trouvaient quelques petites pépites. C'était la majorité des prospecteurs. Toutefois, pour quelques-uns, c'était le jackpot. Le Data mining, c'est la ruée vers l'or transposée aux données.

De nos jours, les entreprises, les administrations, les chercheurs sont inondés de données. À titre d'exemple, si l'on regarde le domaine de la bio-informatique, de 2004 à 2013, donc en moins de dix ans, on est passé de 57 à 4142 génomes bactériens séquencés [1], soit une augmentation de 7266% représentant des To et des To de données. Je ne rentrerai pas ici dans le débat sur l'aspect qualitatif des données (est-ce qu'il vaut mieux avoir peu de données de très bonne qualité, ou beaucoup de données possédant un fort taux d'erreur ?), je m'arrêterai sur un simple fait : il y a trop de données pour pouvoir toutes les analyser manuellement. C'est là que va intervenir le Data mining.

1 Qu'est-ce que le Data mining ?

Littéralement, *Data mining* signifie fouille de données. Le Data mining permet d'explorer et donc, de « fouiller » dans un grand volume de données de manière à les rendre plus compréhensibles et à en extraire des règles de classement et de prédiction. En reprenant le vocabulaire minier, en effectuant un forage dans les données, on doit être en mesure de découvrir des pépites cachées dans le filon des informations.

Le Data mining s'applique toujours sur un important volume de données. Ces données peuvent provenir de sources très différentes : j'ai évoqué précédemment la bio-informatique avec les génomes, mais on peut aussi analyser des données issue de l'astrophysique, des tickets de caisse, etc. Les connaissances extraites de ces données ne seront bien sûr pas toutes exploitées de la même manière :

- certaines permettront de décrire le comportement actuel des données,
- alors que d'autres permettront de prédire leur comportement futur.

Prenons l'exemple des tickets de caisse. Ces petits tickets en savent beaucoup sur votre vie : liste des achats, heure de passage en caisse, moyen de paiement, éventuellement code postal du domicile (voire adresse complète si vous faites livrer vos courses), etc. À partir de l'ensemble des tickets de caisse, on pourrait rechercher des corrélations entre les différents achats. Ce questionnement se fait sans aucun a priori : on ne cherche pas à confirmer une hypothèse, on cherche réellement à extraire de nouvelles connaissances.

L'exemple le plus cité de Data mining concluant (d'un point de vue marketing) est celui de la chaîne de magasins Wal-Mart aux États-Unis. Après analyse de millions de tickets de caisse, il est apparu une corrélation très forte entre l'achat de couches pour bébés et de packs de bière le samedi après-midi. Après analyse, on a pu s'apercevoir que le samedi après-midi les femmes envoyaient leurs maris faire les courses et acheter des couches. Au passage, lesdits maris en profitaient pour s'acheter quelques bières... En plaçant les packs de bière proches des paquets de couches, les ventes ont augmenté ! Vous comprenez peut-être mieux pourquoi quand vous allez au supermarché pour acheter une tablette de chocolat vous vous retrouvez avec un bon de réduction pour un produit qui n'a rien à voir !

La technique mise en œuvre dans cet exemple est l'une des deux techniques de Data mining : la recherche de motifs ou **technique descriptive**. Il ne s'agit que d'une mise en évidence d'informations enfouies dans la base de données. On utilise différentes méthodes statistiques, parmi lesquelles on peut citer : l'ACP (Analyse en Composantes Principales), l'AFC (Analyse Factorielle des Correspondances), le *clustering* ou partitionnement à l'aide d'algorithmes de type k-means, etc.

La seconde technique, la **technique prédictive**, extrapole de nouvelles informations à partir des données de la base. On utilise là encore des méthodes statistiques telles que la régression linéaire, l'ANOVA (*Analysis Of Variance*,

soit analyse de la variance en français), les réseaux de neurones, l'algorithme des k plus proches voisins (dit k-NN), etc.

Toutefois, avant de se lancer dans une fouille de données dans une base, il faut savoir qu'il y a un travail préalable à ne pas négliger : le **nettoyage** des données. C'est de la qualité des données que va dépendre le résultat ! Par exemple, il faut avoir dans la base un nombre suffisant d'observations pour espérer pouvoir en extraire des connaissances, suivant les méthodes employées il faudra s'assurer qu'il n'y a pas de champs incomplets dans les enregistrements, etc.

2 Domaines d'application

D'après un sondage de 2012 du site kdnuggets.com [2], les dix domaines dans lesquels le Data mining est le plus utilisé sont les suivants (les personnes ayant répondu au sondage pouvaient sélectionner plusieurs réponses) :

- 28.6% : CRM (*Customer Relationship Management* ou Gestion de la Relation Client) ;
- 16.3% : Santé ;
- 14.8% : Vente au détail ;
- 14.3% : Banque ;
- 14.3% : Éducation ;
- 13.3% : Publicité ;
- 12.8% : Détection des fraudes ;
- 12.2% : Réseaux sociaux ;
- 11.7% : Sciences ;
- 10.2% : Finance.

Si l'on regroupe les domaines financiers, 81.2% des sondés ont appliqué cette technique dans un but de profit économique (contre 42.3% pour le trio Santé-Éducation-Sciences). Nous étudierons donc deux exemples faisant partie du domaine majoritaire : les CRM et la banque.

2.1 Gestion de la Relation Client

En partant du principe qu'il est plus coûteux d'acquérir un client que de le conserver (démarches commerciales, etc.), le fait de mieux connaître le client et ses habitudes va permettre de lui proposer des produits plus adaptés et d'ainsi augmenter sa satisfaction et par là même, sa fidélité. On peut également prévoir à partir de quel moment il pourrait être à nouveau intéressé par un autre produit.

Cet aspect de service peut être très important si, dans la branche dans laquelle se trouve le fournisseur, les produits ne sont pas clairement différenciés par des avancées technologiques ou par leur prix.

2.2 Banque

Dans le secteur bancaire on utilise le « scoring » pour détecter en amont de l'achat les « bons » clients qui pourraient avoir besoin d'un crédit. En effet, l'objectif de la banque est de vendre le plus de produits financiers possibles et donc d'éviter de se faire doubler par des concurrents éventuellement présents sur le lieu de la transaction (prêt pour l'achat d'un véhicule contracté directement auprès du vendeur, etc.). Mais cet objectif est lié à un autre objectif : la maîtrise des risques et donc de prêter à des clients en capacité de rembourser leur emprunt. En fonction de ces critères, si l'offre de prêt parvient au bon client, au bon moment, la banque aura réussi à atteindre son objectif avec une plus-value : la satisfaction du client et l'enchaînement de celui-ci jusqu'à ce qu'il ait remboursé son prêt.

3 Exemple d'application d'un algorithme de Data mining

Maintenant que nous avons dégrossi la problématique du Data mining, je vous propose d'analyser un algorithme de fouille de données assez simple proposé par Agrawal et Srikant [2] en 1994. Cet algorithme s'appelle Apriori et permet de découvrir des règles d'association. Nous aurons besoin pour le comprendre de quelques définitions préliminaires.

3.1 Définitions

L'algorithme Apriori va analyser un ensemble de **transactions** (modifications de la base de données : une transaction peut contenir plusieurs enregistrements) pour essayer de découvrir des règles indiquant des associations possibles entre différents **items** (les valeurs des enregistrements). Une transaction est un ensemble d'items dans un ordre donné et les items définissent un **domaine** d'application. En reprenant l'exemple des courses dans un supermarché, le domaine pourrait être `D = { Couches, Bière, Chips, Dentifrice }` et nous pourrions avoir trois transactions décrivant le comportement de trois clients du supermarché : `T1 = { Couches, Bière }`, `T2 = { Bière, Couches }`, et `T3 = { Chips, Bière, Dentifrice }`. Notez bien que les transactions `T1` et `T2` sont différentes de par la nature ordonnée des transactions.

Un ensemble d'items est **fréquent** s'il correspond à un motif apparaissant de nombreuses fois dans la base (l'ensemble des transactions). Pour simplifier les exemples, nous allons raisonner en termes de produits étiquetés par des lettres. Ainsi, notre domaine devient `D = { A, B, C, D }`. Si nous avons l'ensemble de transactions `{ ABCD, AB, CAB, DABC, CD }`, alors nous pouvons dire que l'ensemble d'items `AB` est fréquent avec un **support** de `4/5 = 0.8`. Le support est calculé en effectuant la division du nombre d'occurrences du motif par le

cardinal (ou la taille) de l'ensemble des transactions. Pour signifier qu'un ensemble d'items est fréquent, on fixe un seuil minimal de support noté **minSup**. Avec **minSup = 3/5 = 0.6**, on pourrait dire que **AB** est fréquent.

Lorsque nous détecterons des règles, nous leur attribuerons une confiance calculée comme suit : **confiance(X → Y) = support(X U Y) / support(X)**. Dans notre exemple, la règle **AB → C** a une confiance de : **support(AB U C) / support(AB) = 0.4 / 0.8 = 0.5**. Pour savoir si nous retenons une règle ou pas, nous fixerons un seuil de confiance minimal noté **minConf**.

Dans l'algorithme Apriori, nous utiliserons les notations suivantes :

- **Ck** est l'ensemble des items candidats (à être des items fréquents) de longueur **k**. C2 contiendra par exemple les items **{ AB, AC, AD, CA, CB, ... }** ;
- **Lk** est l'ensemble des items fréquents de longueur **k**.

3.2 Apriori

L'algorithme Apriori va comporter deux étapes : la première étape permet de déterminer l'ensemble des items fréquents, puis la seconde étape va générer les règles d'association à partir de l'ensemble des items fréquents. Voici l'algorithme sous forme de pseudo-code :

```
Calculer L1 l'ensemble des items fréquents de longueur 1
k ← 1
Tant que k est différent de Ø (ensemble vide) Faire
      Ck+1 ← items candidats générés depuis Lk
      Pour chaque transaction t dans la base de données Faire
            Incrémenter le compteur de tous les candidats de Ck+1 présents dans t
      Fin Pour
      Lk+1 ← { c Є Ck+1 /  compteur(c) >= minSup }
      k ← k + 1
Fin Tant que
Retourner Uk Lk (union de tous les Lk)
```

Une fois la liste des items fréquents obtenue, on utilisera un autre algorithme pour détecter les règles d'association :

```
Pour chaque item fréquent I Є Uk Lk
      Générer l'ensemble E des sous-ensembles non vides de I
      Pour chaque s Є E
            Si support(I) / support(s) >= minConf Alors
                  La règle s → (I - s) est inférée
            Fin Si
      Fin Pour chaque
Fin Pour chaque
```

Pour comprendre ces algorithmes, prenons un exemple concret sur le domaine **D = { Couches, Bières, Chips, Dentifrice, Sirop }**. Nous disposons de neuf transactions résumées dans le tableau ci-dessous :

Identifiant de transaction	Liste d'items
T1	Couches, Bières, Sirop
T2	Bières, Dentifrice
T3	Bières, Chips
T4	Couches, Bières, Dentifrice
T5	Couches, Chips
T6	Bières, Chips
T7	Couches, Chips
T8	Couches, Bières, Chips, Sirop
T9	Couches, Bières, Chips

Nous allons poser un support minimum de 22% (**minSup = 2/9 = 0.22**) et pour accepter des règles, notre seuil de confiance sera de 70% (**minConf = 0.7**). Nous appliquons alors l'algorithme Apriori :

1ère étape : génération de L1, la liste des items fréquents de longueur 1

Cette étape est assez simple, puisqu'il suffit de calculer le support de chaque item et de ne conserver que ceux pour lesquels celui-ci est supérieur au support minimum. La liste des items candidats **C1** et leur support peut être représentée sous forme de tableau :

Items	Support
{ Couches }	6/9
{ Bières }	7/9
{ Chips }	6/9
{ Dentifrice }	2/9
{ Sirop }	2/9

minSup étant égal à 2/9, **L1** est en fait égal à **C1** puisque tous les items sont conservés. L1 **= { Couches, Bières, Chips, Dentifrice, Sirop }**.

2ème étape : génération de L2, la liste des items fréquents de longueur 2

Nous allons commencer par calculer la liste des items candidats **C2** en partant des combinaisons possibles en utilisant **L1**. Ensuite, nous calculerons le support de l'ensemble des items. Les résultats sont présentés dans le tableau ci-dessous :

Items	Support
{ Couches, Bières }	4/9
{ Couches, Chips }	4/9
{ Couches, Dentifrice }	1/9
{ Couches, Sirop }	2/9
{ Bières, Chips }	4/9
{ Bières, Dentifrice }	2/9
{ Bières, Sirop }	2/9
{ Chips, Dentifrice }	0/9
{ Chips, Sirop }	1/9
{ Dentifrice, Sirop }	0/9

Tous les items dont le support est inférieur à 2/9 ne sont pas conservés. Donc **L2 = { { Couches, Bières }, { Couches, Chips }, { Couches, Sirop }, { Bières, }, { Bières, Dentifrice }, { Bières, Sirop } }**.

3ème étape : génération de L3, la liste des items fréquents de longueur 3

Les opérations réalisées ici sont les mêmes que celles de la deuxième étape : calcul de **C3**, puis déduction de **L3** d'après les valeurs des supports. Ici, il y a une étape supplémentaire de nettoyage des données permettant de détecter les ensembles qui ne pourront pas être fréquents. Je n'ai retenu qu'un seul de ces ensembles dans le tableau ci-dessous, mais **C3** avant nettoyage doit bien sûr contenir l'ensemble des combinaisons issues de **L2**.

Items	Support
{ Couches, Bières, Chips }	**2/9**
{ Couches, Bières, Dentifrice }	Nettoyage : les sous-ensembles sont ici **{ Couches, Bières }**, **{ Couches, Dentifrice }** et **{ Bières, Dentifrice }**. Or, le support de **{ Couches, Dentifrice }** est inférieur à **minSup**. Nous savons donc que l'ensemble de départ ne pourra pas être fréquent.
{ Couches, Bières, Sirop }	**2/9**

Donc **L3 = { Couches, Bières, Chips }, { Couches, Bières, Sirop } }**.

4ème étape : génération de L4, la liste des items fréquents de longueur 4

Après nettoyage, **C4** est égal à l'ensemble vide et l'algorithme s'achève alors ici. Maintenant, les ensembles **L1**, **L2**, et **L3** vont être utilisés pour détecter les règles d'association.

Dernière étape : inférence des règles d'association

Ici, il faut passer au second algorithme. Il faut parcourir l'ensemble des sous-ensembles d'items fréquents de **{ L1, L2, L3 } = { { Couches }, ..., { Couches, Bières, Chips }, { Couches, Bières, Sirop } }**. Comme tout cela va faire beaucoup de sous-ensembles, je vais me restreindre à appliquer l'algorithme au dernier sous-ensemble **{ Couches, Bières, Sirop }**. L'ensemble E des sous-ensembles issus de cet ensemble sont **{ { Couches, Bières }, { Couches, Sirop }, { Bières, Sirop }, { Couches }, { Bières }, { Sirop } }**. Nous avions choisi pour seuil de confiance la valeur 70%. Nous pouvons alors générer les règles et déterminer celles qui seront conservées :

- **R1 : { Couches, Bières } → { Sirop }**

  D'où vient cette règle ? Si nous reprenons les notations de l'algorithme, **I = { Couches, Bières, Sirop }**. Nous prenons un ensemble **s** dans E. Ici, **s = { Couches, Bières }**. La règle générée est donc **s → (I - s)** soit **{ Couches, Bières } → ({ Couches, Bières, Sirop } - { Couches, Bières })**.

Il reste à tester le seuil de confiance (voir les calculs de support dans les tableaux précédents) : **support(I) / support(s) = 2/9 / 4/9 = 0.5**. Comme **minConf = 0.7**, cette règle est rejetée.

- **R2 : { Couches, Sirop } → { Bières }**

  Le calcul de confiance donne **2/9 / 2/9 = 1**, donc cette règle est conservée.

- **R3 : { Bières, Sirop } → { Couches }**

  La règle est conservée avec une confiance de 100%.

- **R4 : { Couches } → { Bières, Sirop }**

  La règle est rejetée avec une confiance de 33%.

- **R5 : { Bières } → { Couches, Sirop }**

  La règle est rejetée avec une confiance de 29%.

- **R6 : { Sirop } → { Couches, Bières }**

  La règle est conservée avec une confiance de 100%.

En ne déroulant qu'une partie de notre exemple, nous avons déjà trouvé trois règles d'association fortes **R2**, **R3** et **R6** illustrant parfaitement le cas des magasins Wal-Mart...

Conclusion

Nous n'avons fait qu'effleurer le domaine du Data mining dans cet article. Il existe de nombreux algorithmes différents pour extraire des connaissances nouvelles des données. Bien sûr, il ne faut pas forcément coder ces algorithmes, car il existe de nombreux logiciels permettant de faire du Data mining tels que Weka, ou encore Tanagra. Mais malgré toutes ces techniques et ces outils, il ne faut surtout pas oublier un aspect essentiel : pour « faire parler » des données, il faut avoir des données... et de bonne qualité ! ■

Bibliographie

[1] Base de données de génomes : http://www.genomesonline.org/cgi-bin/GOLD/index.cgi

[2] Sondage 2012 sur les domaines d'application du Data mining : http://www.kdnuggets.com/polls/2012/where-applied-analytics-data-mining.html

[3] Agrawal (Rakesh) et Srikant (Ramakrishnan), « Fast Algorithms for Mining Association Rules », Proceedings VLDB, 1994.

CONCEPTUALISATION D'UNE BASE DE DONNÉES AVEC LA MÉTHODE MERISE

par Tristan Colombo

Vous avez certainement dû entendre parler de Merise ou UML, mais ne savez peut-être pas à quoi correspondent ces termes, ni comment les utiliser ? Dans cet article, nous découvrirons les principes fondamentaux de la méthode Merise permettant de réfléchir à la conception d'une base de données.

Lorsque l'on développe un logiciel utilisant une base de données, il faut créer des tables qui vont contenir les informations et éventuellement être en relation les unes avec les autres. Si vous travaillez sur un micro-projet ne contenant que deux ou trois tables, vous pourrez vous lancer directement dans la création de votre base. Si vous avez besoin d'une dizaine de tables et que vous manipulez régulièrement des bases de données, là encore vous pourrez vous lancer directement dans la mise en œuvre de votre base. Dans ces deux cas, si votre base n'évolue pas par la suite, vous ne devriez pas rencontrer de problème. Par contre, si cette dernière est susceptible d'être modifiée (en général, on ne le sait malheureusement pas à l'avance...), ou qu'elle comporte de nombreuses tables, il vous faudra passer par une étape préalable de réflexion pour organiser les données :

- Quelles tables créer ?
- Comment organiser les données au sein de chaque table (colonnes) ?
- Quelles seront les relations présentes entre certaines tables ?

Il s'agit des fondations du projet, qui ont les mêmes fonctions que les fondations d'une maison : en cas de problème, tout s'écroule ! Bien sûr, on peut trouver des personnes utilisant de nombreux étais pour empêcher la destruction totale de la maison, et en informatique, on peut rafistoler... jusqu'au point de rupture. J'ai travaillé dans une entreprise où la base de données contenait plus de cinquante tables et où aucun schéma n'avait jamais été réalisé. Lorsque l'on a essayé de l'obtenir par *reverse engineering* et travail manuel, on s'est aperçu que c'était impossible : le modèle n'était plus relationnel...

La phase de conception est donc très importante, si ce n'est essentielle, pour pouvoir utiliser une base de données de manière pérenne. Il existe plusieurs méthodes permettant d'aider le développeur à atteindre son objectif, Merise est l'une d'elles.

1 Présentation générale de Merise

Merise est une méthode conçue dans les années 1970 et rendue opérationnelle dans les années 1980 par Hubert Tardieu et Arnold Rochfeld **[I-3]**. Elle est basée sur une approche systémique, c'est-à-dire en considérant le projet comme un système possédant des flux de données entrants et sortants.

Merise n'est pas la seule méthode systémique, on peut citer SAGACE, AXIAL, MEGA, etc. On retrouve la théorie de Merise enfouie dans les logiciels de conception de base de données, tels que MySQL Workbench et autres, que nous développerons dans un article de ce hors-série. Donc, même si par

la suite nous n'utilisons pas tout le formalisme de Merise, comprendre son fonctionnement nous aidera à utiliser les logiciels de conception de base de données.

Merise se base sur une organisation en 4 niveaux :

- le niveau **conceptuel** : conception du système d'information en faisant abstraction des contraintes techniques et organisationnelles ;
- le niveau **organisationnel** : organisation des ressources humaines et matérielles ;
- le niveau **logique** : organisation logique du système en faisant abstraction des caractéristiques techniques précises ;
- le niveau **physique** : organisation du système d'un point de vue technique.

Dans cet article, nous nous concentrerons sur la conceptualisation des données au détriment du traitement. Nous cherchons pour l'instant simplement à concevoir une base pour stocker nos données. Si la base est bien conçue, manipuler des données de cette base se fera simplement. Si nous commençons à structurer notre base en fonction des traitements à venir, nous risquons fort d'obtenir rapidement un schéma aberrant. La partie « traitement » de Merise est une réflexion sur les traitements globaux de l'application et c'est donc cette partie qui ne sera pas traitée ici. Nous nous concentrerons sur les niveaux conceptuel, logique et physique, en abordant le **modèle conceptuel de données** (ou MCD), le **modèle logique de données** (ou MLD) et enfin, le **modèle physique de données** (ou MPD).

Merise est plus particulièrement utilisé en France et en Europe. Il existe peu de logiciels de modélisation Merise et encore moins de logiciels libres. Pour réaliser cet article, j'ai utilisé AnalyseSI qui est une application Java. Il vous suffit de télécharger le fichier **analyseSI-0.75.jar** sur https://launchpad.net/analysesi/+download ou de récupérer le code source avec **git** depuis https://github.com/AnalyseSI/AnalyseSI.git. Si votre installation de Java est complète, vous lancerez AnalyseSI en exécutant :

```
java -jar analyseSI-0.75.jar
```

Vous obtiendrez alors l'ouverture d'une fenêtre semblable à celle présentée en figure 1.

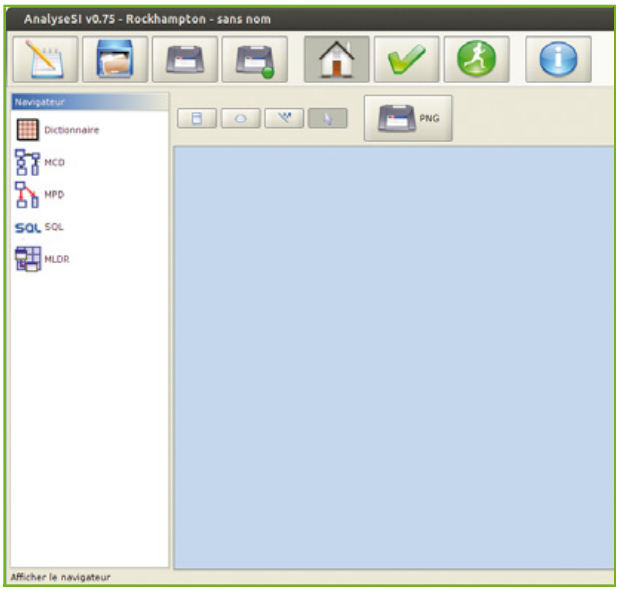


Fig. 1 : Interface du logiciel AnalyseSI

2 Le Modèle Conceptuel de Données (MCD)

Le MCD constitue la première étape de la réflexion sur la conception de la base de données. Dans cette étape, nous prendrons pour exemple un client qui souhaite acheter des articles sur un site marchand. Ici nous faisons complètement abstraction des contraintes techniques et nous ne réaliserons notre schéma qu'à partir des éléments permettant de représenter nos informations.

Commençons par les clients. Un client va être défini par un certain nombre d'informations : un prénom, un nom, une adresse postale, etc. Mais ces éléments, qui sont appelés **propriétés** de l'**entité** Client, ne suffisent pas à l'identifier de manière unique : il nous faut une **clé** nommée également **identifiant**. Ici, la clé sera le numéro de client. D'un point de vue schématique, l'identifiant sera souligné.

Avec les informations dont nous disposons, nous pouvons déjà modéliser l'entité client comme le montre la figure 2. Chaque propriété a un type particulier. Ce n'est pas dans le MCD que l'on doit préciser le type, puisque celui-ci se trouve au niveau de la définition physique du modèle... Mais si vous utilisez AnalyseSi, vous remarquerez que dès la création du MCD ces informations vous sont demandées (voir figure 2). Je rappelle donc que ce n'est pas l'objectif de cette étape.

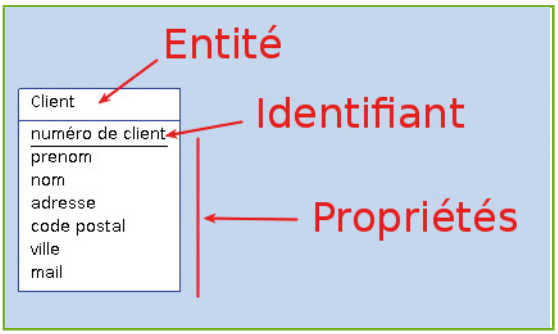


Fig. 2 : Schéma de l'entité Client

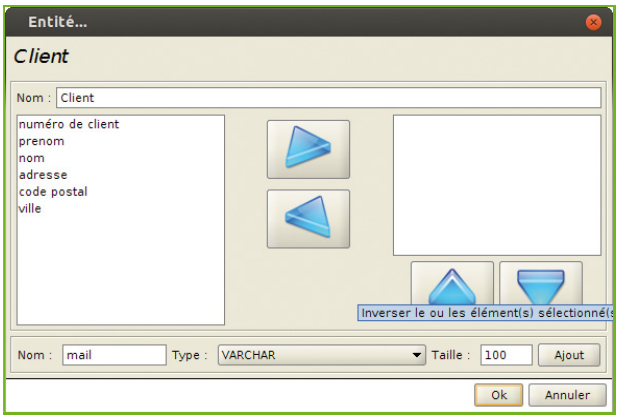


Fig. 3 : Création des propriétés liées à une entité dans AnalyseSI

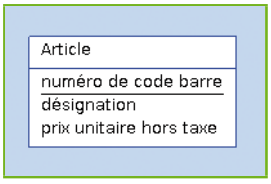


Fig. 4 : Schéma de l'entité Article

Le client va pouvoir acheter des articles. Un article possède lui aussi un certain nombre de propriétés, dont un identifiant, le tout rattaché à l'entité Article. Nous pourrons associer à cette entité un numéro de code-barres, une désignation et un prix unitaire hors taxe. La figure 4 montre le schéma de cette entité qui est créée de la même manière que l'entité Client.

Dans notre exemple, un client peut commander un article. Il existe donc une **relation** (ou encore une **association**) entre les entités Client et Article. Cette relation sera formalisée par un verbe à l'infinitif décrivant l'action à effectuer entre les deux entités et elle sera représentée par un ovale relié aux deux entités, comme le montre la figure 5. La relation pourra être lue de gauche à droite « un client commande un article » ou de droite à gauche « un article est commandé par un client ».

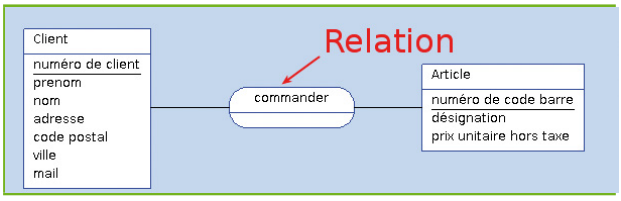


Fig. 5 : Schéma de l'exemple Client – commander – Article

Sur le schéma, sous le verbe « commander », on peut voir un trait délimitant deux espaces, comme sur la représentation des entités. Tout comme une entité peut contenir des propriétés, une relation peut elle aussi contenir des propriétés. À ce moment-là, on dit que la relation est **porteuse**. Dans notre exemple, nous pourrions par exemple vouloir autoriser la commande de plusieurs articles identiques en une seule saisie. Il faut alors ajouter la propriété « quantité » à la relation « commander » (voir figure 6).

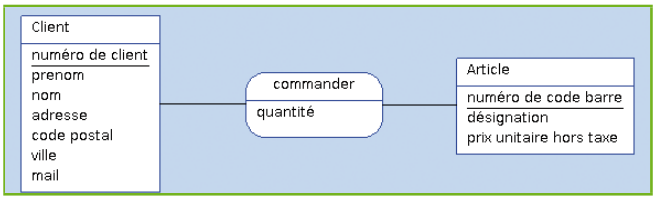


Fig. 6 : Schéma de l'exemple Client – commander – Article avec une relation porteuse

Nous avons ici affaire à une relation binaire. Il est possible d'obtenir des relations ternaires ou quaternaires en connectant trois ou quatre entités. Dans ce cas, il faudra s'interroger sur la cohérence du schéma : faut-il lier une nouvelle entité à la relation, ou créer une nouvelle propriété dans la relation ? Par exemple, si nous souhaitons ajouter la date d'une commande, faut-il créer une relation ternaire entre les entités Client, Article et Date, ou ajouter simplement une propriété « date » dans la relation « commander » ? Les deux solutions sont valides. Posez-vous simplement la question de la clarté de votre schéma et choisissez la formulation la plus lisible.

Nous arrivons maintenant à une étape qui peut paraître anecdotique, mais qui est très importante pour la définition précise des données qui seront stockées dans notre base : la **cardinalité**. La cardinalité indique le nombre de fois où une entité participe à une relation. On note la **cardinalité minimale**, qui exprime le nombre minimum d'occurrences et la **cardinalité maximale**, qui exprime le nombre maximum d'occurrences en les séparant par une virgule. Si nous reprenons notre exemple, il faut nous poser les questions suivantes :

1. Combien de clients commandent des articles ?
2. Combien d'articles peuvent être commandés par chaque client ?

Pour chacune des questions, nous devons déterminer la cardinalité minimale et maximale. Ainsi, pour la première question, au minimum, combien de clients peuvent commander des articles ? Est-ce que zéro est une valeur raisonnable ? Si je n'ai pas de client, je n'ai pas de commande... Donc, au minimum, il nous faut un client.

Pour le maximum, soit pour des raisons particulières nous souhaitons imposer un quota maximum de commandes par client, soit - ce qui paraît le plus logique - chaque client peut commander autant d'articles qu'il le souhaite. La cardinalité maximale sera alors notée **n**.

En ce qui concerne la seconde question, le raisonnement est similaire : si un client ne commande pas d'article, c'est que la commande n'a pas lieu et un client peut commander autant d'articles qu'il le souhaite. La cardinalité sera donc la même que précédemment : 1, n où 1 indique la cardinalité minimale et n indique la cardinalité maximale. La figure 7 montre l'application de cette cardinalité sur notre schéma. Les cardinalités les plus courantes sont : (0, 1), (0, n), (1, 1), et (1, n).

Abonnez-vous !

Profitez de nos offres d'abonnement spéciales disponibles au verso !

Téléphonez au 03 67 10 00 20 ou commandez par le Web

Économisez plus de 25%*

* Sur le prix de vente unitaire France Métropolitaine

11 Numéros de GNU/Linux Magazine

Les 3 bonnes raisons de vous abonner :

- Ne manquez plus aucun numéro.
- Recevez GNU/Linux Magazine chaque mois chez vous ou dans votre entreprise.
- Économisez 24,50 €/an ! (soit plus de 3 magazines offerts !)

4 façons de commander facilement :

- par courrier postal en nous renvoyant le bon ci-dessous
- par le Web, sur www.ed-diamond.com
- par téléphone, entre 9h-12h et 14h-18h au 03 67 10 00 20
- par fax au 03 67 10 00 21

ABONNEMENT

11 GNU/LINUX MAGAZINE / FRANCE

58€*

au lieu de 82,50 €* en kiosque

Économie : 24,50 €*

*OFFRE VALABLE UNIQUEMENT EN FRANCE MÉTROPOLITAINE
Pour les tarifs hors France Métropolitaine, consultez notre site : www.ed-diamond.com

Bon d'abonnement à découper et à renvoyer à l'adresse ci-dessous

Tournez SVP pour découvrir toutes les offres d'abonnement >>

Voici mes coordonnées postales :	
Société :	
Nom :	
Prénom :	
Adresse :	
Code Postal :	
Ville :	
Pays :	
Téléphone :	
e-mail :	

❑ Je souhaite recevoir les offres promotionnelles et newsletter des Éditions Diamond.

❑ Je souhaite recevoir les offres promotionnelles de nos partenaires.

GNU/LINUX MAGAZINE / FRANCE

Édité par Les Éditions Diamond
Service des Abonnements
B.P. 20142 - 67603 Sélestat Cedex
Tél. : + 33 (0) 3 67 10 00 20
Fax : + 33 (0) 3 67 10 00 21

Vos remarques :

.....

.....

En envoyant ce bon de commande, je reconnais avoir pris connaissance des conditions générales de vente des Éditions Diamond à l'adresse internet suivante : **www.ed-diamond.com/cgv** et reconnais que ces conditions de vente me sont opposables.

Tournez SVP pour découvrir toutes les offres d'abonnement

PROFITEZ DE NOS OFFRES D'ABONNEMENT SPÉCIALES POUR LIRE PLUS ET FAIRE DES ÉCONOMIES !

➡ Abonnement

offre 1 — ABONNEMENT — 11 LINUX MAGAZINE / FRANCE — **58€*** — au lieu de 82,50€** en kiosque — Economie : 24,50 €

Vous pouvez également vous abonner sur :

www.ed-diamond.com

ou par Tél. : +33 (0)3 67 10 00 20 /

Fax : +33 (0)3 67 10 00 21

NOUVEAU !

numerique.ed-diamond.com

pour vous abonner et acheter vos magazines en format numérique (PDF)

unixgarden.com

pour retrouver une sélection d'articles des Éditions Diamond

* Tarifs France Métro (F)

** Base tarifs kiosque zone France Métro (F)

➡ Voici nos offres d'abonnements groupés incluant GLMF

offre 3 — ABONNEMENTS GROUPÉS — 11 LINUX MAGAZINE / FRANCE + 6 LINUX PRATIQUE — **85€*** — au lieu de 121,50€** en kiosque — Economie : 36,50 €

offre 4 — ABONNEMENTS GROUPÉS — 11 LINUX MAGAZINE / FRANCE + 6 Hors-séries — **89€*** — au lieu de 130,50€** en kiosque — Economie : 41,50 €

offre 5 — ABONNEMENTS GROUPÉS — 11 LINUX MAGAZINE / FRANCE + 6 MISC — **90€*** — au lieu de 133,50€** en kiosque — Economie : 43,50 €

offre 6 — ABONNEMENTS GROUPÉS — 11 LINUX MAGAZINE / FRANCE +6 HS + 6 LINUX PRATIQUE — **119€*** — au lieu de 169,50€** en kiosque — Economie : 50,50 €

offre 7 — ABONNEMENTS GROUPÉS — 11 LINUX MAGAZINE / FRANCE +6 HS + 6 MISC — **124€*** — au lieu de 181,50€** en kiosque — Economie : 57,50 €

offre 8 — ABONNEMENTS GROUPÉS — 11 LINUX MAGAZINE / FRANCE +6 HS + 6 LINUX PRATIQUE + 6 MISC — **154€*** — au lieu de 220,50€** en kiosque — Economie : 66,50 €

offre 9 — ABONNEMENTS GROUPÉS — 11 LINUX MAGAZINE / FRANCE +6 HS + 6 LINUX PRATIQUE + 6 MISC + 6 LINUX ESSENTIEL — **184€*** — au lieu de 259,50€** en kiosque — Economie : 75,50 €

offre 12 — ABONNEMENTS GROUPÉS — 11 LINUX MAGAZINE / FRANCE +6 HS + 6 LINUX PRATIQUE +3 HS + 6 MISC +2 HS + 6 LINUX ESSENTIEL — **215€*** — au lieu de 301,50€** en kiosque — Economie : 86,50 €

offre 2 — ABONNEMENTS GROUPÉS — 6 LINUX ESSENTIEL + 6 LINUX PRATIQUE — **60€*** — au lieu de 78,00€** en kiosque — Economie : 18,00 €

➡ Voici nos autres offres d'abonnements groupés

offre 10 — ABONNEMENTS GROUPÉS — 6 MISC + 2 Hors-séries — **48€*** — au lieu de 69,00€** en kiosque — Economie : 21,00 €

offre 11 — ABONNEMENTS GROUPÉS — 6 LINUX PRATIQUE + 3 Hors-séries — **48€*** — au lieu de 63,00€** en kiosque — Economie : 15,00 €

offre 15 — ABONNEMENTS GROUPÉS — 6 LINUX ESSENTIEL + 6 LINUX PRATIQUE +3 HS — **78€*** — au lieu de 102,00€** en kiosque — Economie : 24,00 €

➡ Nos Tarifs s'entendent TTC et en euros

		F France Métro	D DOM	T TOM	Zone 1 Europe	Zone 2 Afrique / Orient	Zone 3 Amérique	Zone 4 Asie / Océanie
1	Abonnement GLMF	58 €	78 €	94 €	80 €	87 €	84	80 €
2	Abonnement LPE + LP	60 €	86 €	105 €	88 €	96 €	92 €	89 €
3	Abonnement GLMF + LP	85 €	120 €	145 €	123 €	134 €	129 €	124 €
4	Abonnement GLMF + GLMF HS	89 €	122 €	147 €	125 €	136 €	131 €	126 €
5	Abonnement GLMF + MISC	90 €	128 €	151 €	130 €	141 €	136 €	131 €
6	Abonnement GLMF + GLMF HS + Linux Pratique	119 €	164 €	198 €	168 €	183 €	176 €	170 €
7	Abonnement GLMF + GLMF HS + MISC	124 €	172 €	204 €	175 €	190 €	183 €	177 €
8	Abonnement GLMF + GLMF HS + MISC + LP	154 €	214 €	255 €	218 €	237 €	228 €	221 €
9	Abonnement GLMF + GLMF HS + MISC + LP + LPE	184 €	258 €	309 €	263 €	286 €	275 €	266 €
10	Abonnement MISC + MISC HS	48 €	66 €	76 €	66 €	72 €	69 €	68 €
11	Abonnement LP + LP HS	48 €	65 €	78 €	66 €	72 €	70 €	68 €
12	Abonnement GLMF + GLMF HS + MISC + MISC HS + LP + LP HS + LPE	215 €	297 €	355 €	302 €	329 €	317 €	307 €
15	Abonnement LPE + LP + LP HS	78 €	109 €	132 €	111 €	121 €	117 €	113 €

- ZONE 1 : Allemagne, Belgique, Danemark, Italie, Luxembourg, Norvège, Pays-Bas, Portugal, Suède, Autriche, Espagne, Finlande, Grande Bretagne, Grèce, Islande, Suisse, Irlande, Estonie, Croatie, Slovenie, Slovaquie, Republique Tcheque, Pologne, Bielorussie, Bosnie Herzegovine, Bulgarie, Chypre, Georgie, Hongrie, Lettonie, Lituanie, Macedoine, Malte, Moldova, Roumanie, Russie, Serbie, Ukraine, Albanie, Armenie, ...
- ZONE 2 : Algerie, Maroc, Tunisie, Turquie, Afrique du Sud, Seychelle, Senegal, Israel, Palestine, Syrie, Jordanie, Botswana, Cameroun, Cap Vert, Comores, Rep.Dom. Congo, Cote d'Ivoire, Egypte, Kenya, Libye, Madagascar, Nigeria, ...
- ZONE 3 : Canada, Etats Unis, Guyana, Haiti, Republique Dominicaine, Jamaique, Argentine, Bresil, Cuba, Mexique, ...
- ZONE 4 : Australie, Japon, Chine, Coree du Nord, Coree du Sud, Inde, Indonesie, Nouvelle Zelande, Taiwan, Thailande, Vietnam, ...

Mes choix :

Mon 1er choix	Je sélectionne le N° (1 à 15) de l'offre choisie :	
Mon 2ème choix	Je sélectionne le N° (1 à 15) de l'offre choisie :	
Mon 3ème choix	Je sélectionne le N° (1 à 15) de l'offre choisie :	
	Je sélectionne ma zone géographique (F à Zone 4) :	
	J'indique la somme due : (Total)	€

Exemple : je souhaite m'abonner à l'offre GNU/Linux Magazine + GNU/Linux Magazine Hors-série + MISC (offre 7) et je vis en Belgique (zone 1), ma référence est donc 7zone1 et le montant de l'abonnement est de 175 euros.

Je choisis de régler par :

- ❑ Chèque bancaire ou postal à l'ordre des Éditions Diamond
- ❑ Carte bancaire n° ⌊_⌊_⌊_⌊_⌋ ⌊_⌊_⌊_⌊_⌋ ⌊_⌊_⌊_⌊_⌋ ⌊_⌊_⌊_⌊_⌋
- Expire le : ⌊_⌊_⌊_⌊_⌋
- Cryptogramme visuel : ⌊_⌊_⌊_⌋

Date et signature obligatoire

LINUX MAGAZINE / FRANCE

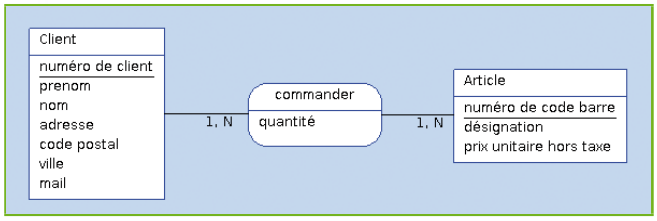


Fig. 7 : Schéma de l'exemple Client – commander – Article avec notation des cardinalités

Une fois le Modèle Conceptuel de Données réalisé, on passe au Modèle Logique de Données.

3 Le Modèle Logique de Données (MLD)

L'objectif du MLD est de partir du MCD et de nous rapprocher le plus possible du modèle physique, sans s'encombrer des contraintes techniques. Nous allons suivre trois règles qui vont nous permettre de supprimer les relations pour les intégrer dans une entité ou les transformer en entités. Pourquoi ? Tout simplement parce que ce que nous stockons dans une base de données, ce sont des tables (donc des entités) et non des relations.

Voici maintenant chacune des règles dépendant des cardinalités appliquées à la relation.

3.1 Cardinalité (0, n) / (1, 1), (0, n) / (0, 1), (1, n) / (1, 1) ou (1, n) / (0, 1)

La transformation du MCD en MLD va supprimer la relation dont les propriétés vont être absorbées par la table issue de l'entité ayant la cardinalité la plus faible ((0, 1) ou (1, 1)). Un lien entre les deux tables sera créé à l'aide d'une clé étrangère qui référencera la clé primaire de la table de plus forte cardinalité (voir figure 8).

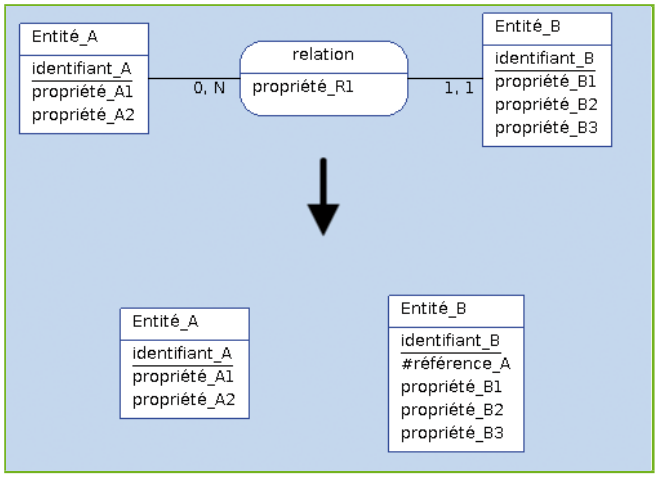


Fig. 8 : Transformation MCD vers MLD d'une relation (0, n) / (1, 1), (0, n) / (0, 1), (1, n) / (1, 1) ou (1, n) / (0, 1)

3.2 Cardinalité (0, n) / (0, n), (1, n) / (0, n) ou (1, n) / (1, n)

Dans ce cas, la relation va se transformer en table qui conservera les mêmes propriétés, mais dans laquelle la clé primaire sera composée de deux clés étrangères qui référenceront les deux clés primaires des tables intervenant dans la relation. La figure 9 illustre ce mécanisme.

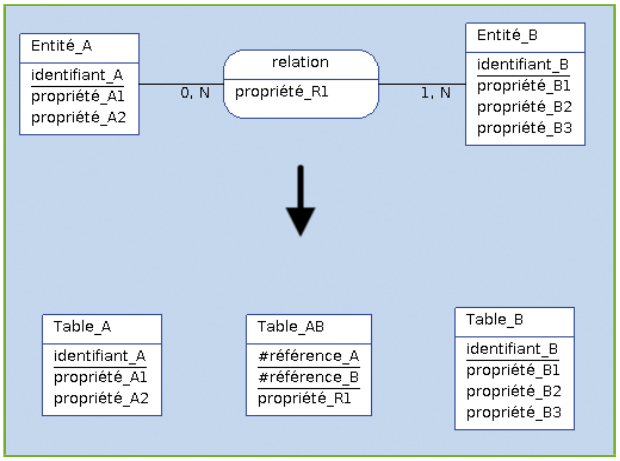


Fig. 9 : Transformation MCD vers MLD d'une relation (0, n) / (0, n), (1, n) / (0, n) ou (1, n) / (1, n)

3.3 Cardinalité (1, 1) / (1, 1)

Il s'agit ici d'un cas particulier : on va regrouper les deux entités et la relation au sein d'une seule et même table qui contiendra l'ensemble des propriétés comme le montre la figure 10.

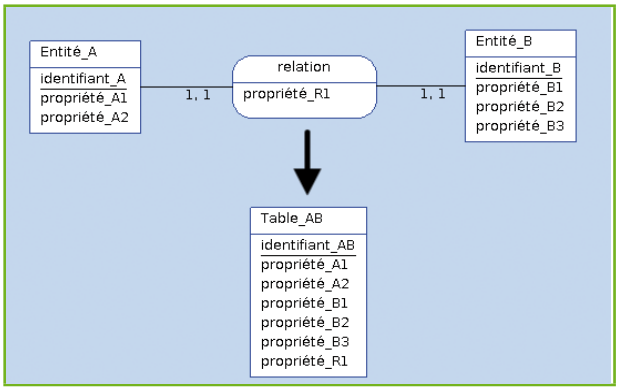


Fig. 10 : Transformation MCD vers MLD d'une relation (1, 1) / (1, 1)

3.4 Et sur l'exemple de départ ?

En appliquant ces règles sur notre exemple, nous nous retrouvons dans le cas exposé en 3.2 et nous obtiendrons alors trois tables, comme le montre la figure 11, page suivante.

Une fois le Modèle Logique de Données obtenu, il faut créer le Modèle Physique de Données.

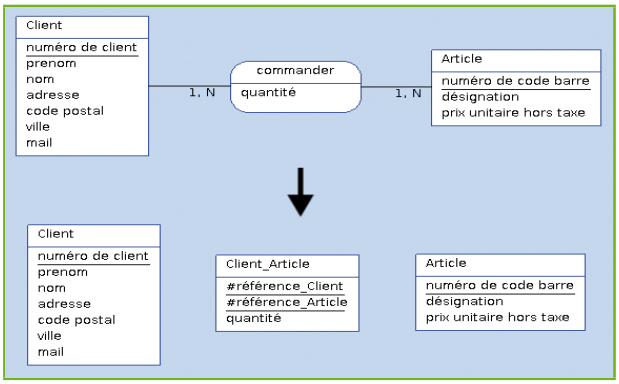


Fig. 11 : Transformation MCD vers MLD de notre exemple Client – commander – Article

4 Le Modèle Physique de Données (MPD)

La traduction du MLD en MPD n'est en fait qu'un passage d'une représentation graphique à une représentation textuelle, dans laquelle les considérations techniques interviendront : quel est le type des données, peut-on optimiser les requêtes en ajoutant des index ou des colonnes pré-calculées, etc. ? En général, lors de cette étape on passe directement au langage de requête du système de gestion de bases de données que l'on souhaite employer (le plus souvent SQL).

Pour rester généraliste, la transformation du MLD au MPD de notre exemple Client - commande - Article en pseudo-SQL donnerait :

```
CRÉER LA TABLE Client
{
  numero_de_client : ENTIER NON NULL EN INCRÉMENTATION AUTOMATIQUE (CLÉ PRIMAIRE),
  prenom : CHAÎNE DE CARACTÈRES,
  nom : CHAÎNE DE CARACTÈRES,
  adresse : CHAÎNE DE CARACTÈRES,
  code_postal : ENTIER,
  ville : CHAÎNE DE CARACTÈRES,
  mail : CHAÎNE DE CARACTÈRES
}

CRÉER LA TABLE Article
{
  numero_de_code_barre : CHAÎNE DE CARACTÈRES NON NULLE EN INCRÉMENTATION
AUTOMATIQUE (CLÉ PRIMAIRE),
  désignation : CHAÎNE DE CARACTÈRES,
  prix_unitaire_hors_taxe : RÉEL
}

CRÉER LA TABLE commander
{
  FK_commander_numero_de_client :  ENTIER NON NULL RÉFÉRENCE Client.numero_de_
client (CLÉ ÉTRANGÈRE),
  FK_numero_de_code_barre :  CHAÎNE DE CARACTÈRES NON NULLE RÉFÉRENCE Article.
numero_de_code_barre (CLÉ ÉTRANGÈRE),
  quantité : ENTIER,
  CLÉ_PRIMAIRE -> FK_commander_numero_de_client,  FK_numero_de_code_barre
}
```

À partir du MCD, AnalyseSI est capable de générer le MPD et le code SQL de création des tables (cliquez sur le bonhomme qui court dans un bouton vert).

5 Et UML dans tout ça ?

Comme son nom l'indique, le langage de modélisation UML – pour *Unified Modeling Language* ou Langage de Modélisation Unifié, permet de modéliser des projets. Il propose treize diagrammes permettant d'analyser un projet dans toutes ses étapes depuis le diagramme des cas d'utilisation, décorrélé de toute considération technique, jusqu'au diagramme de classes s'approchant grandement du MCD. Merise est une approche beaucoup plus théorique qu'UML, qui est orienté vers la conception (orientée objet qui plus est).

Alors pourquoi choisir l'un ou l'autre ? Techniquement, Merise et UML sont complémentaires : Merise est plus particulièrement adapté aux systèmes d'information organisationnels (analyse métier), alors qu'UML est adapté au système d'information informatisé (application informatique associée). Dans un projet, selon le positionnement des acteurs, qu'il soit analyste, développeur, consultant ou chef de projet, le choix se portera préférentiellement sur l'une ou l'autre des méthodes. Malgré la forte ressemblance entre le MCD de Merise et le diagramme de classes d'UML, dans le cadre de la modélisation d'une base de données, la méthode la plus adaptée reste Merise.

Conclusion

Merise est une méthode adaptée à la conceptualisation complète d'un projet. Nous n'avons vu ici qu'une petite partie de la méthode qui nous servira à comprendre d'autres logiciels de conceptualisation concentrés uniquement sur la base de données. Que ce soit Merise ou UML, si l'on respecte toutes les étapes, cela va prendre énormément de temps. Ce sont des méthodes qui sont clairement orientées pour de gros projets avec des équipes de développement conséquentes.

Pour un développement en effectif très réduit, il faut se restreindre à ce qui constituera le cœur du projet. Non pas que les autres étapes doivent être complètement oubliées, mais leur réalisation pourra être faite de manière plus rapide. Pour un unique développeur travaillant sur un projet, il ne pourra pas passer des mois en analyse et en modélisation, il faudra trouver un compromis permettant de partir d'une base saine et de réajuster constamment le projet. On passe alors sur des méthodes dites « agiles », de type *Extreme Programming*. ■

Bibliographie

[1] TARDIEU H., ROCHFELD A., COLETTI R., « La méthode Merise, tome 1 : Principes et outils », éditions d'Organisation, 1983.

[2] TARDIEU H., ROCHFELD A., COLETTI R., PANET G., VAHEE G., « La méthode Merise, tome 2 : Démarche et pratiques », éditions d'Organisation, 1985.

[3] ROCHFELD A., MOREJON J., « La méthode Merise, tome 3 : Gamme opératoire », éditions d'Organisation, 1989.

À NE PAS RATER !

OPEN SILICIUM N° 7

DISPONIBLE CHEZ VOTRE MARCHAND DE JOURNAUX ET SUR :

www.ed-diamond.com

LES SYSTÈMES DE GESTION DE BASES DE DONNÉES LIBRES

par Jean-Michel Armand

Devoir utiliser un Oracle coûtant un bras, voire deux, en licences et autant en équipement serveur n'est pas une obligation, même lorsque l'on veut mettre en place de gros systèmes. Il y a en effet plusieurs systèmes de base de données libres plus que performants. Nous allons vous les présenter dans cet article.

1 MySQL, le plus commun

MySQL **[1]** est sans aucun doute le système de gestion de base de données le plus couramment utilisé. Il a pour lui deux avantages : il se configure très facilement et ses performances ne sont pas mauvaises. Originellement, MySQL était développé par une société suédoise, MySQL AB, qui publia la première version en 1995. Actuellement, les dernières versions stables sont la 5.5.31 et la 5.6.11. MySQL est publié sous double licence, à savoir la GPL et une licence propriétaire.

L'histoire de MySQL depuis quelques années n'a pas été sans cahot et sans crainte pour sa survie. En effet, en 2008, MySQL AB a été rachetée par Sun. Parmi les impacts positifs de ce rachat, cela a permis que MySQL devienne le moteur de base de données à utiliser avec Java. Par la suite, en 2009, Oracle a racheté Sun et de ce fait, MySQL. Ce rachat a fait peser les plus grandes craintes pour le futur de MySQL, craintes qui ont conduit quelques utilisateurs à se tourner vers d'autres SGBD.

Michael Widenuis, le fondateur de MySQL AB, a quant à lui décidé de forker MySQL pour créer MariaDB, qui vous sera présenté plus loin dans cet article.

1.1 Installation

La solution la plus simple consiste à l'installer en utilisant le gestionnaire de paquets de votre distribution favorite.

```
~$ sudo aptitude install mysql-server mysql-common mysql-client
```

Oubli ou changement de mot de passe

Il peut arriver que vous oubliiez le mot de passe administrateur de votre base de données. Comment faire dans ce cas là ? Il y a deux solutions :

- soit en tapant (si vous êtes sous Debian-like) :

```
sudo service mysql reset-password
```

- soit en demandant à **dpkg** de relancer la configuration du serveur en lançant la commande suivante :

```
sudo dpkg-reconfigure mysql-server-5.5
```

Si vous voulez simplement changer votre mot de passe, il suffira de taper :

```
sudo mysqladmin -u root password NewPassWord -p OldPassWord
```

Si vous n'aviez pas donné de mot de passe à l'installation, pour en ajouter un vous devez taper :

```
sudo mysqladmin -u root password Nouveau_mot_de_passe -p ""
```

Lors de l'installation, on va vous demander le mot de passe administrateur pour MySQL. C'est la seule chose que vous devrez configurer à ce moment-là. Une fois l'installation finie, votre MySQL fonctionne.

Pour tester votre MySQL, il suffit d'essayer de se loguer avec le client en ligne de commandes. Pour ce faire, tapez :

```
~$ mysql -uroot -p
```

Si tout s'est bien passé, MySQL devrait vous demander votre mot de passe et ensuite vous serez connecté.

1.2 Configuration

La configuration de MySQL est principalement rassemblée dans le fichier **/etc/my.cnf**. La variable **socket** vous permettra de configurer la socket utilisée par MySQL. **basedir**, **datadir** et **tempdir** vous permettront de modifier les dossiers d'installation, de données et de stockage des fichiers temporaires. **key_buffer_size** vous permettra de configurer la taille du *buffer* utilisé pour le cache des blocs d'index.

Pour activer le log des requêtes les plus lentes, il vous suffira de donner **1** comme valeur à la variable **slow_query_log** et la variable **log-slow-queries** vous permettra d'indiquer le nom du fichier où stocker les informations en question. La variable **max_connections** permet de configurer le nombre maximum de clients MySQL simultanés.

Pour finir, deux variables de configuration du cache : **query_cache_limit** permet de définir à partir de quelle taille maximum les résultats ne sont plus mis en cache et **query_cache_size** permet de définir la taille de la mémoire allouée pour la mise en cache des requêtes précédentes.

Une dernière chose concernant la configuration de MySQL : je vous conseille vraiment d'utiliser InnoDB comme moteur de base de données. Si vous suivez mon conseil, il faut que vous sachiez que par défaut, MySQL stocke toutes les informations nécessaires au bon fonctionnement de vos tables en InnoDB dans un seul fichier qui grossit continuellement. Plusieurs possibilités existent pour modifier ce comportement. L'une d'entre elles consiste à demander à MySQL de stocker les informations table à table en utilisant un fichier différent pour chaque table. Pour cela, il vous suffit d'ajouter la ligne **innodb_file_per_table** dans la section **[mysqld]** du fichier de configuration **my.cnf**.

MyISAM et InnoDB

MySQL gère plusieurs types de formats de base de données (plus d'une vingtaine). Lorsque vous créez une table, vous pouvez choisir lequel de ces formats (on parle aussi de moteurs) sera utilisé pour sa gestion. MyISAM et InnoDB sont les plus connus :

- MyISAM est le moteur le plus simple de MySQL. Il ne gère ni les transactions, ni les clés étrangères. Il est par contre relativement rapide et léger ;
- InnoDB est un moteur bien plus robuste que MyISAM. Il gère les transactions et les clés étrangères, mais cette plus grande robustesse le conduit à faire gonfler la taille des bases qui l'utilisent comme moteur.

1.3 Outils en ligne de commandes

MySQL possède plusieurs outils en ligne de commandes. **mysqladmin**, nous l'avons déjà vu, permet de changer les mots de passe des utilisateurs. Grâce à lui, vous pourrez aussi créer ou supprimer directement vos bases, exporter des tables ou changer les droits d'accès. **mysqldump** vous permettra d'exporter dans un fichier texte le contenu d'une base (ou de toutes vos bases) :

```
~$ mysqldump --opt BD_NAME > FILE_NAME
~$ mysqldump --all-databases > FILE_NAME
```

L'option **--opt** est souvent oubliée et elle est pourtant fondamentale. C'est en effet une option raccourcie qui vous ajoutera d'un coup toutes les options suivantes: **--add-drop-table --add-locks --create-options --disable-keys --extended-insert --lock-tables –quick** et **--set-charset**.

mysqlimport vous permettra d'importer des fichiers texte dans une table.

mysqldumpslow vous affichera le journal des requêtes les plus lentes si vous avez activé l'option **log-slow-query** sur le serveur.

Enfin, deux outils de maintenance et de vérification sont fournis par MySQL :

- Le premier, **myisamchk**, ne fonctionne que pour les tables au format MyISAM. Il vous permettra de vérifier, optimiser et même réparer les tables utilisant ce format ;
- Le second, **mysqlcheck**, fonctionne sur tous les formats. Lui aussi vous permettra d'analyser les tables et vous pourrez aussi les réparer ou tenter de les optimiser.

Attention : pour ces deux derniers outils, le serveur doit être arrêté.

Le client **mysql** en ligne de commandes offre quelques commandes d'administration, en plus de vous permettre d'interagir en SQL avec votre base. Une fois connecté à votre base, vous allez pouvoir être informé du statut du serveur avec **SHOW STATUS ;**.

SHOW VARIABLES ; et **SHOW VARIABLES LIKE 'log %' ;** vous permettront d'afficher les variables de configuration

du serveur. Enfin, **`SHOW PROCESSLIST ;`** vous permettra de lister tous les processus qui tournent sur votre serveur MySQL et **`KILL Numero_Processus ;`** de les tuer.

1.4 Outils web et graphiques

Il existe principalement deux outils pour interagir avec MySQL : phpMyAdmin, qui est un outil web, et MySQL Workbench vu dans l'article sur la modélisation des bases de données.

1.4.1 PhpMyAdmin

PhpMyAdmin est une application bien connue des développeurs web. Vous avez deux possibilités pour l'installer : soit utiliser votre gestionnaire de paquets préféré, soit la télécharger sur son site **[2]** et la déployer comme n'importe quelle application web. Personnellement, si votre distribution est assez réactive sur les mises à jour de sécurité, je vous conseille d'utiliser votre gestionnaire de paquets :

```
~$ sudo aptitude install phpmyadmin
```

PhpMyAdmin est livrée avec une configuration par défaut. Suivant votre distribution et votre gestionnaire de paquets, cette configuration par défaut sera directement dans **`/etc/apache2/conf.d`**. Si ce n'est pas le cas, elle se trouvera dans **`/etc/phpmyadmin/apache.conf`**. Il vous suffira alors d'un lien pour ajouter cette configuration dans le répertoire de configuration d'Apache et d'un **`reload`** pour pouvoir profiter de phpMyAdmin :

```
~# ln -s /etc/phpmyadmin/apache.conf /etc/apache2/conf.d/phpmyadmin
~# /etc/init.d/apache2 reload
```

Une fois cela fait, vous n'avez plus qu'à vous rendre avec votre navigateur à l'adresse : http://IP_De_Votre_Machine/phpmyadmin. La figure 1 montre une capture d'une session phpMyAdmin.

Attention : la configuration proposée par phpMyAdmin est une configuration bien trop ouverte pour que vous puissiez la laisser en production ! Pensez au moins à changer l'alias concernant l'URL de phpMyAdmin, cela évitera que vous ne subissiez tous les robots tentant d'exploiter les failles de phpMyAdmin. N'oubliez pas également de limiter soit par authentification, soit par IP, les connexions possibles.

2 MariaDB, le petit nouveau

MariaDB **[3]** a été créé par réaction au rachat de Sun par Oracle... C'est donc un SGBD tout récent. C'est un fork de MySQL qui a pour parti pris de rester compatible à 100% avec MySQL. MariaDB est publié sous licence GPL.

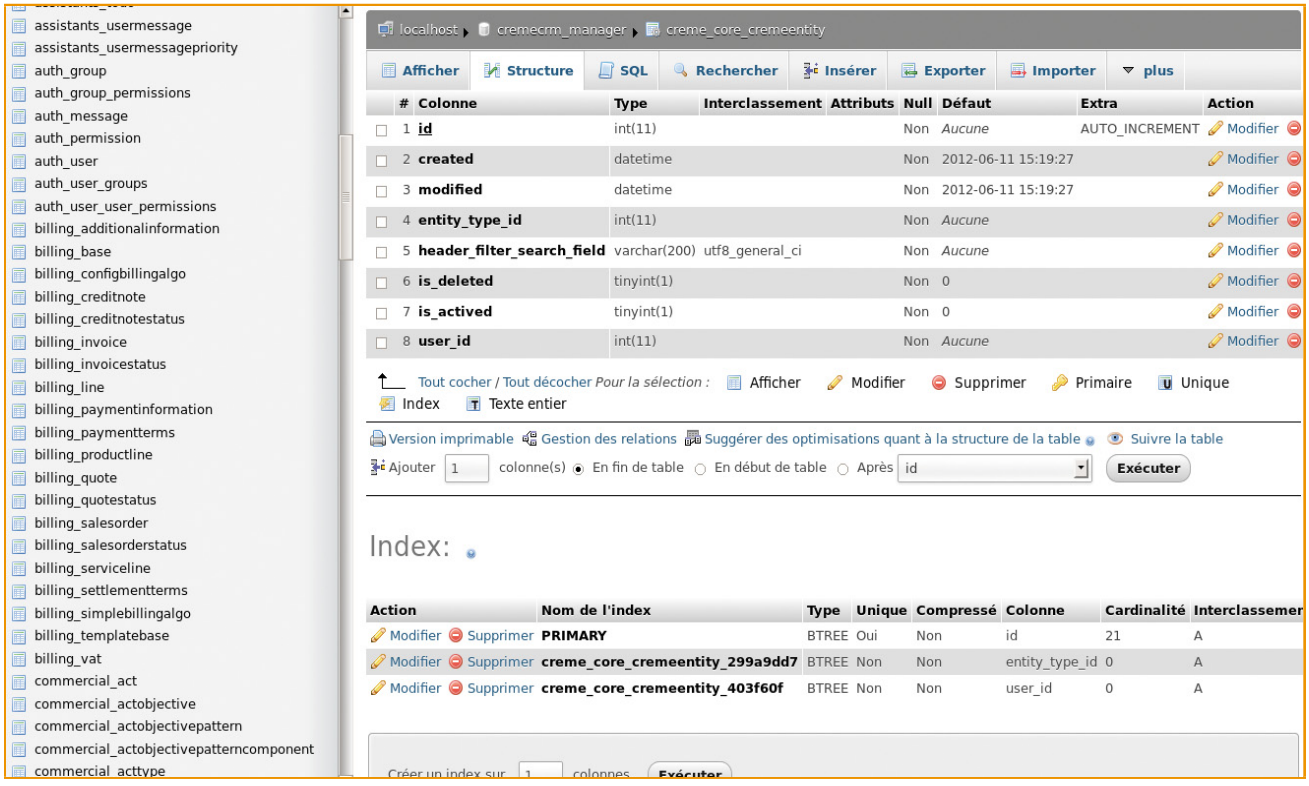


Fig. 1 : La structure d'une table vue par phpMyAdmin

Depuis quelques mois (avril 2013), la société derrière MariaDB a signé un accord avec la société SkySQL, afin de développer MariaDB dans une vision « NewSQL », qui viserait à regrouper le meilleur du SQL et du NoSQL.

Le fait que MariaDB soit un fork de MySQL permet que les données gérées par MySQL le soient aussi par MariaDB. Vous pouvez donc choisir de migrer de MySQL à MariaDB sans problème, même si vous avez déjà des données (c'est d'ailleurs ce qu'a fait l'encyclopédie Wikipédia). Par contre, faites attention : l'installation de MariaDB, lorsque vous la faites avec les paquetages, entraîne la désinstallation de MySQL !

2.1 Installation

MariaDB fournit des dépôts pour la plupart des distributions Linux (une page **[4]** de son site web en donne la liste). Si vous êtes sous Ubuntu, vous n'aurez qu'à taper les lignes suivantes :

```
~$ sudo apt-key adv --recv-keys --keyserver keyserver.ubuntu.com 1BB943DB
~$ echo deb http://ftp.igh.cnrs.fr/pub/mariadb//repo/5.5/ubuntu $(lsb_release -sc) main |
sudo tee /etc/apt/sources.list.d/MariaDB.list
~$ sudo aptitude update
~$ sudo aptitude install mariadb-server mariadb-client
```

Si, par contre, vous n'utilisez pas l'une des distributions des dépôts, il vous faudra compiler MariaDB. Pour cela, vous aurez besoin d'installer **cmake**, **bzr**, **gcc** et **g++**, les **autoconfs**, **mysql** et quelques autres dépendances. Heureusement, la documentation est très bien faite et il vous suffira de vous laisser guider, étape par étape.

Lors de l'installation, tout comme pour celle de MySQL, il vous sera demandé un mot de passe pour le compte administrateur de MariaDB.

2.2 Configuration

Première chose : les moteurs de gestion des données sont différents entre MySQL et MariaDB. MyISAM est remplacé par Aria et InnoDB est remplacé par XtraDB. Pour le reste, là encore, la configuration se fait dans le fichier **my.cnf** présent dans **/etc/mysql**.

2.3 Outils

En fait, les outils sont exactement les mêmes que pour MySQL. Si vous voulez vous connecter en ligne de commandes, il vous suffira de taper :

```
~$ mysql -p
```

La seule différence sera le message d'accueil, que vous pourrez lire alors, ainsi que le prompt qui s'affichera :

```
MariaDB [(none)]>
```

De la même manière, phpMyAdmin ou MySQL Workbench fonctionneront parfaitement avec MariaDB.

3 PostgreSQL, le plus performant

PostgreSQL **[5]** est publié sous la licence PostgreSQL qui est très similaire à une licence BSD. Sa première version date de la même année que celle de MySQL, à savoir 1995. Sa dernière version est la 9.2 et la 9.3 est en bêta. PostgreSQL se distingue par son importante robustesse, sa puissance en termes de réplication et ses très nombreuses extensions. PostgreSQL est ainsi capable de gérer aussi bien des données géolocalisées que de mettre en place dans ses tables des champs ressemblant à du NoSQL, qui permettent de stocker des documents sans schéma et d'ensuite faire des recherches dans les données qu'ils contiennent.

3.1 Installation

L'installation se fait tout simplement grâce à votre gestionnaire de paquets. Si vous êtes sous Debian-like, un simple **aptitude** suffira :

```
~$ sudo aptitude install postgresql
postgresql-client
```

3.2 Configuration

Une fois l'installation réalisée, la première chose à faire sera de vous créer un nouvel utilisateur. En effet, PostgreSQL se configure normalement avec l'utilisateur système **postgres** qui est le seul habilité à lancer la commande **psql** qui permet de réaliser les tâches d'administration sur PostgreSQL (vous pouvez faire le test en tentant de lancer **psql** avec votre utilisateur classique). Mais cet utilisateur **postgres** est un peu comme le **root** sous GNU/Linux : on ne doit pas l'utiliser pour les tâches courantes. Nous allons donc tout de suite créer ce nouvel utilisateur :

```
~$ sudo -i -u postgres
~$ createuser -P NOM_USER --createdb
--createrole
```

Et voilà, vous avez un nouvel utilisateur que vous pourrez utiliser pour vos tâches courantes.

Pour le reste, la configuration de votre PostgreSQL se fait dans le fichier **/etc/postgresql/NUMERO_VERSION/main/postgresql.conf**. Quelques valeurs sont facilement modifiables pour améliorer les performances de votre base de données. La variable **shared_buffers** configure la taille

de mémoire que le serveur va utiliser en tant que mémoire partagée. Par défaut, elle est à 24Mo. Vous pouvez l'augmenter sans problème : si votre serveur de base de données est seul sur un serveur dédié, vous pouvez attribuer 25% de la RAM pour cette mémoire partagée. Si vous voulez configurer la taille de RAM qu'utiliseront les workers pour les opérations de tri interne ou de table de hachage, c'est la variable **work_mem** qu'il faudra modifier. La valeur par défaut est de 1Mo : vous pouvez l'augmenter entre 10Mo et 100Mo.

La variable **checkpoint_segments** modélise le nombre maximum de journaux de transaction qui seront stockés entre deux points de vérification. La valeur par défaut est de 3 : il peut être bon de l'augmenter légèrement à 6 ou 10. Par contre, l'augmentation de cette valeur allongera le délai de remise en service en cas d'arrêt brutal du serveur.

Enfin, la variable **maintenance_work_mem** gère la quantité de mémoire que pourront utiliser les opérations de maintenance comme **VACUUM, CREATE INDEX** ou la création de clé étrangère. Une valeur importante à ce niveau-là peut améliorer la rapidité des opérations **VACUUM** ou celles de restauration des sauvegardes. Une bonne fourchette de valeur serait entre 15 et 25% de la taille de la RAM, sur un serveur dédié à PostgreSQL.

3.3 Outils web et graphiques

Toute comme pour MySQL, il existe différents logiciels pour se connecter à PostgreSQL.

3.3.1 PhpPgAdmin

PhpPgAdmin est l'interface web qui permet de se connecter à PostgreSQL (voir figure 2). Il s'installe aussi simplement que phpMyAdmin, grâce à votre outil de gestion de paquets :

```
~$ sudo aptitude install phppgadmin
```

Il va installer son fichier de configuration dans **/etc/phppgadmin** et si cela n'est pas fait automatiquement, vous devrez mettre un lien symbolique sur le fichier **/etc/phppgadmin/apache.conf** pour que celui-ci soit lu par Apache. Là encore, la même recommandation que pour phpMyAdmin : reprenez la configuration Apache pour la sécuriser un peu plus.

Si vous tentez d'utiliser phpPgAdmin avec l'utilisateur **postgres**, celui-ci refusera de vous connecter en vous expliquant que cette interdiction est là pour des raisons de sécurité. En effet, tout comme un serveur SSH ne devrait jamais être en **PermitRootLogin true**, phpPgAdmin vous interdit d'utiliser le compte **root** de PostgreSQL pour vous loguer. Vous pouvez toutefois contourner cette sécurité en mettant à **false** la variable **$conf['extra_login_security']** dans le fichier **/usr/share/phppgadmin/conf/config.inc.php**.

3.3.2 PgAdmin3

PgAdmin3 est un client lourd, sous licence PostgreSQL, qui vous permettra de vous connecter à différents serveurs PostgreSQL (voir figure 3). L'installation est aussi simple que l'installation de n'importe quel paquet pour votre distribution :

Fig. 2 : Interface de phpPgAdmin

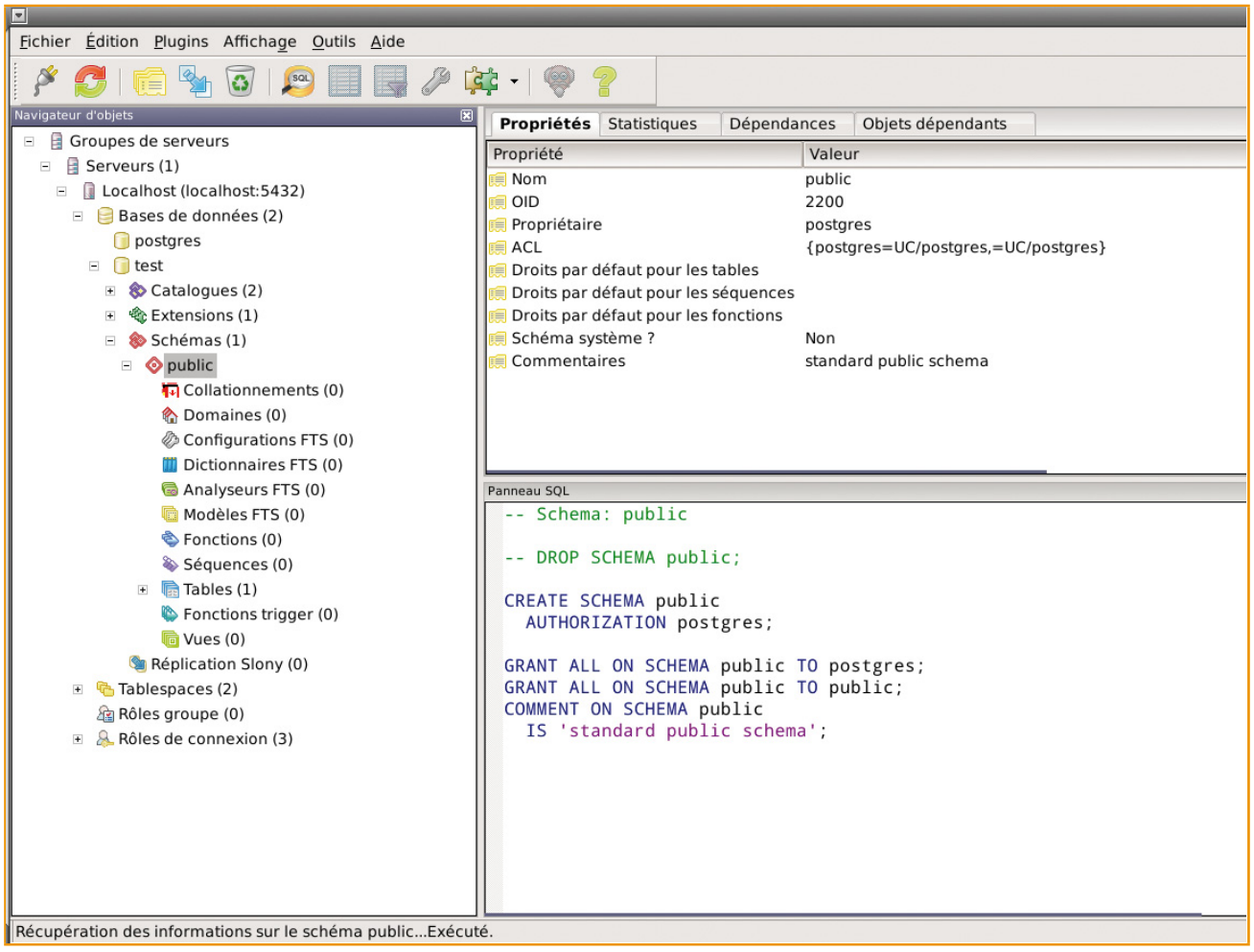


Fig. 3 : Interface de pgAdmin3

```
~$ sudo aptitutde install pgadmin3
```

En plus de pouvoir taper vos scripts SQL, les exécuter, créer et modifier des bases, vous pourrez aussi directement modifier les fichiers de configuration et obtenir une console psql.

Conclusion

Si vous utilisez MySQL, l'une des questions à se poser est de savoir s'il faut migrer vers MariaDB. Bien que MariaDB semble encore un peu jeune, il semblerait toutefois que de plus en plus d'arguments poussent en faveur de la migration. Sans même parler du fait que la vie de MySQL est suspendue au bon vouloir d'Oracle ! Mais en restant à un niveau strictement technique, il semblerait bien que, tant au niveau des performances que de la réplication ou de la sécurité, MariaDB soit en avance comparé à MySQL.

Pourtant, avant d'enterrer complètement l'ancêtre MySQL, il faudra attendre de voir comment se passent les migrations de ceux qui ont déjà osé sauter le pas. Pour le reste, lorsque vos besoins sont importants et nécessitent robustesse et performance, voire besoin fonctionnel spécifique, là, le choix n'est pas très difficile à faire : PostgreSQL est très souvent la bonne solution, même s'il vous demandera bien plus d'investissement en termes de réglage et de configuration. ■

Références

[1] MySQL : https://www.mysql.fr/

[2] phpMyAdmin : http://www.phpmyadmin.net/home_page/index.php

[3] MariaDB : https://mariadb.org/

[4] Dépôt MariaDB : https://downloads.mariadb.org/mariadb/repositories/

[5] PostgreSQL : http://www.postgresql.org/

MODÉLISATION DE LA BASE DE DONNÉES :

UN SCHÉMA POUR SAVOIR OÙ L'ON VA, DES OUTILS POUR GÉRER LES DONNÉES

par Tristan Colombo

Nous avons vu que grâce à des méthodes telles que Merise nous pouvions réfléchir au schéma de notre base de données en suivant trois phases correspondant aux trois modèles : MCD, MLD et MPD. Ce formalisme a été utilisé par des éditeurs de logiciels permettant d'interagir avec les systèmes de gestion de bases de données. Les outils qu'ils nous proposent permettent de créer simplement un schéma de base de données et de l'appliquer pour donner naissance à une base de données.

Dans cet article, nous nous concentrerons sur deux outils disponibles pour les deux plus gros systèmes de gestion de bases de données libres : **MySQL** (ou MariaDB) et **PostgreSQL**. Pour MySQL (ou MariaDB), il s'agit de MySQL Workbench et pour PostgreSQL, il s'agit de SQL Power Architect. Si vous avez besoin d'installer l'un ou l'autre de ces systèmes de gestion de bases de données, vous pouvez vous reporter à l'article précédent portant sur les SGBD libres. Nous considérerons ici que vous disposez d'une installation fonctionnelle de ces systèmes.

1 MySQL Workbench

Pour installer MySQL Workbench, il vous suffit de vous rendre sur la page http://dev.mysql.com/downloads/tools/workbench/#downloads et de sélectionner la distribution pour laquelle vous souhaitez obtenir un paquetage. À défaut, vous pourrez sélectionner le code source. Je vais choisir ici le paquetage Debian associé à la distribution Ubuntu. Pour l'installer, il suffit d'ouvrir un terminal et d'exécuter :

```
sudo dpkg -i mysql-workbench-gpl-5.2.47-1ubu1204-amd64.deb
```

En cas d'erreur, lisez bien les messages de log qui vous indiquent quelles sont les librairies manquantes que vous pourrez installer par un simple **sudo aptitude install nom_librairie**. Chez moi, il a fallu en ajouter quatre :

```
sudo aptitude install libctemplate0 libzip2 python-paramiko python-pysqlite2
```

Vous pouvez aussi vous satisfaire d'une version un peu plus ancienne (la version 5.2.38, alors que la version actuelle est 5.2.47), mais disponible dans les dépôts Ubuntu :

```
sudo aptitude install mysql-workbench
```

Après différents tests, la version 5.2.38 installée depuis les dépôts est beaucoup plus stable.

Vous pourrez lancer MySQL Workbench en tapant dans un terminal **mysql-workbench**. Vous obtiendrez alors l'écran de menu initial de la figure 1. Vous constaterez que l'application permet de faire beaucoup plus que la réalisation d'un schéma, puisqu'elle propose en fait trois outils, présentés dans trois colonnes distinctes :

- *SQL Development* (cadre 1) : permet d'explorer des bases de données et d'effectuer des requêtes. Pour cet outil, nous verrons simplement comment créer une connexion à un serveur de base de données. La partie SQL ne sera pas traitée, car elle sera vue dans l'article suivant.
- *Data Modeling* (cadre 2) : création de schémas de base de données. Des trois outils de MySQL Workbench, c'est le plus important. C'est grâce à lui que nous allons pouvoir créer nos bases de données en suivant plus ou moins les mêmes étapes que la méthode Merise.
- *Server Administration* (cadre 3): administration d'un serveur de base de données. Cet outil permet d'accéder à un serveur de base de données MySQL ou MariaDB pour l'administrer depuis une interface graphique.

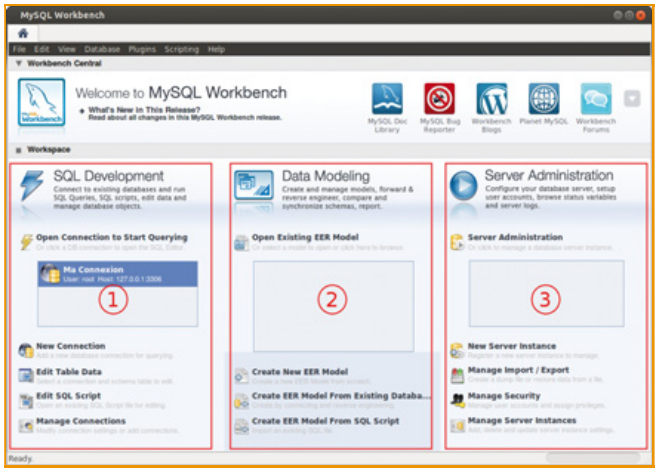


Fig. 1 : Menu principal de MySQL Workbench comportant les trois outils : (1) SQL Development, (2) Data Modeling, et (3) Server Administration.

1.1 SQL Development

La création d'une connexion à un serveur de base de données se fait en cliquant sur le bouton ***New Connection*** de la colonne dédiée à cet outil. Une nouvelle fenêtre apparaît (Fig. 2). Vous devrez donner un nom à votre connexion, ce qui vous permettra de la voir apparaître dans la liste des connexions possibles sur l'interface générale de MySQL Workbench et vous pourrez également créer plusieurs connexions vers différents serveurs de base de données ou avec différents utilisateurs.

Il faudra ensuite indiquer l'adresse IP, ainsi que le port de votre serveur MySQL/MariaDB et le nom de l'utilisateur sous lequel vous souhaitez vous connecter. Vous pouvez décider de stocker son mot de passe en cliquant sur le bouton ***Store in Keychain*** sinon celui-ci vous sera demandé à chaque connexion (avec la possibilité à ce moment-là de le sauvegarder). Enfin, vous pouvez indiquer un schéma par défaut : il s'agit de la base qui sera ouverte automatiquement à la connexion. Si vous laissez l'emplacement vide, aucune base ne sera ouverte.

En cliquant sur le bouton ***Test Connection*** vous pourrez vous assurer que vos paramètres sont corrects avant de valider en cliquant cette fois-ci sur le bouton ***OK***.

Si vous avez sécurisé la connexion à votre serveur par SSH ou SSL, vous pourrez effectuer les configurations nécessaires en cliquant sur l'onglet ***Advanced*** ou en modifiant la méthode de connexion (en haut de la fenêtre).

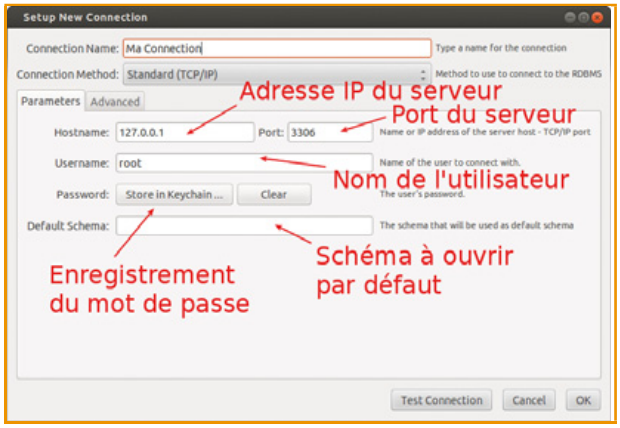


Fig. 2 : Fenêtre de création d'une nouvelle connexion dans SQL Development

Une fois la connexion créée, vous retournez sur la page de MySQL Workbench. En cliquant sur la connexion dans la liste ***Open Connection to Start Querying***, vous aboutirez sur la page visible en figure 3. Dans la colonne de gauche, vous verrez apparaître la liste des bases accessibles. En cliquant sur l'une d'elles, vous pourrez afficher la liste de ses tables, vues et routines. En cliquant à nouveau sur chacun de ces objets vous obtiendrez de nouveaux détails : nom des colonnes pour les tables, etc. Cet outil va vous permettre de créer de nouvelles bases (appelées « schema » dans MySQL Workbench) et de nouvelles tables, vues ou routines depuis la barre d'outils. L'onglet ouvert par défaut (appelé normalement ***SQL File 1***) permet d'écrire une requête SQL et de l'exécuter.

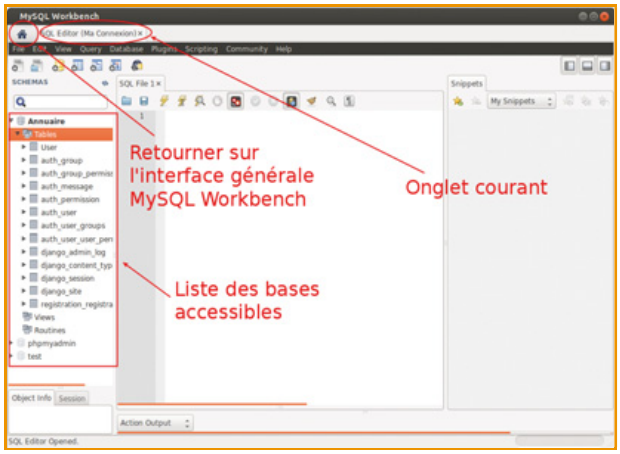


Fig. 3 : Onglet SQL Development dans MySQL Workbench

Notez que SQL Development s'ouvre à l'intérieur de MySQL Workbench sous la forme d'un onglet. Pour retourner sur la page de l'interface générale de MySQL Workbench, il faut cliquer sur le bouton ***Accueil*** (en forme de maison) en haut et à gauche.

1.2 Data Modeling

Dans la colonne ***Data Modeling*** de l'interface principale, vous trouverez quatre boutons :

- ***Open Existing EER Model*** pour ouvrir un schéma existant. Pour l'instant nous n'en avons pas...
- ***Create New EER Model*** pour créer un nouveau schéma. C'est par là que nous allons commencer.
- ***Create EER Model From Existing Database*** et ***Create EER Model From SQL Script*** qui permettent de faire du *reverse engineering* en créant le schéma d'une base depuis une base existante ou son code SQL.

Nous allons donc débuter par la création d'un EER Model... Mais, au fait, de quoi s'agit-il ? Un ER Model (la même chose qu'EER Model mais avec un E en moins) est un **modèle entité-association** (ER pour *Entity-Relationship*) tel que nous l'avons défini dans les articles précédents. Un modèle EER est un **modèle entité-association avancé** (*Enhanced Entity-Relationship*), qui permet de définir plus précisément des schémas plus complexes. Pour créer notre modèle EER, il faut donc cliquer sur le bouton ***Create New EER Model***, ce qui a pour effet d'ouvrir un nouvel onglet (comme pour l'outil SQL Development). Dans cet onglet (Fig. 4), on peut définir cinq zones :

1. Zone de description de l'élément sélectionné. Par exemple, lorsque nous aurons créé une table, et en la sélectionnant, son nom ainsi que sa description s'afficheront dans cette zone.
2. Liste des types utilisateur disponibles pour les champs. Il s'agit en fait de raccourcis faisant référence à une configuration précise des types classiques. Par exemple, un BOOLEAN est défini comme un TINYINT(1). Pour définir vous-même un type particulier, il faudra aller dans le menu ***Model > User Defined Types***.
3. Zone d'ajout et d'ouverture des schémas. Une fois que nous aurons créé des tables, nous utiliserons cette zone pour afficher le schéma.
4. Zone permettant de créer les différents objets de la base : tables, vues et routines. Dans le cadre de cet article, nous nous cantonnerons à la notion de table.
5. Enfin, la dernière zone correspond à des tâches que l'on pourrait qualifier d' « administration » avec la gestion des utilisateurs/rôles, des scripts SQL et d'éventuelles notes sur le modèle.

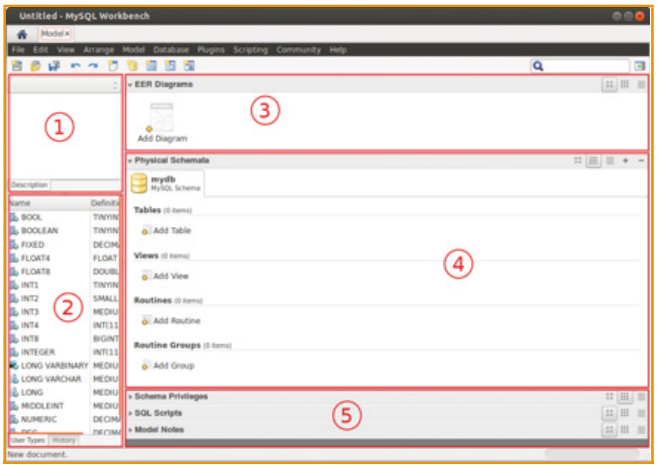


Fig. 4 : Écran principal de l'outil Data Modeling

Pour changer le nom de la base dont le nom par défaut est « maBase », il faut double-cliquer sur l'onglet ***maBase*** en haut et à gauche de la zone 4 (ou alors effectuer un clic droit et sélectionner ***Edit schema***). La modification s'effectuera en bas de la fenêtre dans le champ ***Name***.

Nous allons maintenant suivre les étapes permettant de créer une base de données : conception des tables, création du schéma et création de la base proprement dite.

1.2.1 Ajouter des tables

La phase de conception des tables est elle-même composée de deux sous-phases : la réflexion et la création « physique » de la table. La réflexion s'effectue sur papier et permet de dégager les grands axes du schéma. Nous allons reproduire ici le MCD que nous avions obtenu à l'aide de Merise (Fig. 5).

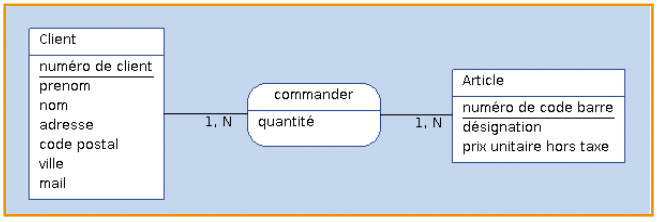


Fig. 5 : MCD de la base de données que l'on souhaite créer

Dans l'outil ***Data Modeling***, on raisonne immédiatement en termes de base et de tables. Nous allons donc commencer par ajouter une table en double-cliquant sur le bouton ***Add Table*** (zone 4). Un nouvel onglet va apparaître en bas de la fenêtre, contenant lui-même plusieurs onglets. Je ne détaillerai pas tous les champs, ni tous les onglets possibles, mais on peut saisir le nom de la table, le moteur de base de données employé pour la base (choix impactant le mode de fonctionnement complet de la base car, par exemple, MyISAM fonctionne en autocommit – ou validation automatique des modifications), la description de la table et bien sûr, dans l'onglet ***Columns***, la liste des champs de la table (Fig. 6).

Sur chaque ligne des champs se trouvent des cases à cocher situées dans des colonnes portant des abréviations étranges... Ces cases permettent en fait d'indiquer :

- ***PK*** : *Primary Key* ou clé primaire ;
- ***NN*** : *Not null* ou non vide. Le champ doit obligatoirement être rempli ;
- ***UQ*** : *UniQue* ou... unique. La valeur de ce champ doit être distincte des valeurs des champs déjà enregistrés ;
- ***ZF***: *Zero Fill* ou remplissage de zéros. Cette option ajoutera, dans le cas de champs numériques, des zéros pour compléter les espaces vides. Par exemple, pour un champ de type INT(10), une insertion de la valeur 12 provoquera en fait l'insertion de 0000000012. Par défaut, un champ marqué ***ZF*** devient non signé (donc attention aux opérations...) ;
- ***AI***: *AutoIncrement* ou incrémentation automatique. Lors de l'ajout d'un élément dans la table, il est inutile de spécifier ce champ qui sera complété automatiquement en utilisant un compteur mis à jour après chaque ajout.

Le dernier champ, ***Default***, contient la valeur à assigner par défaut au champ si celui-ci n'est pas renseigné lors de l'insertion.

Lors de cette saisie, inutile de représenter les relations entre tables, nous pourrons faire cela plus simplement directement sur le schéma. De plus, il n'y a pas d'action particulière à effectuer pour valider les ajouts ou modifications effectués sur une table, ceux-ci sont immédiatement pris en compte.

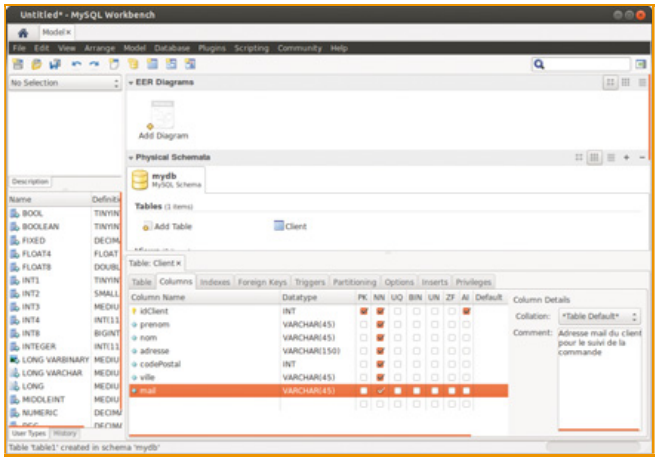


Fig. 6 : Création d'une table dans l'outil Data Modeling de MySQL Workbench

En appliquant la même procédure, nous pouvons créer les tables **`Commande`** et **`Article`** manquantes, toujours sans se soucier des clés étrangères. Notez que, contrairement aux versions antérieures de MySQL Workbench, si vous souhaitez éditer plusieurs tables en même temps, une ouverture simple de la table ne suffira pas car l'onglet courant sera écrasé par le nouvel onglet. Vous devrez effectuer un clic droit sur la table et sélectionner ***Edit in new Window*** (Fig. 7).

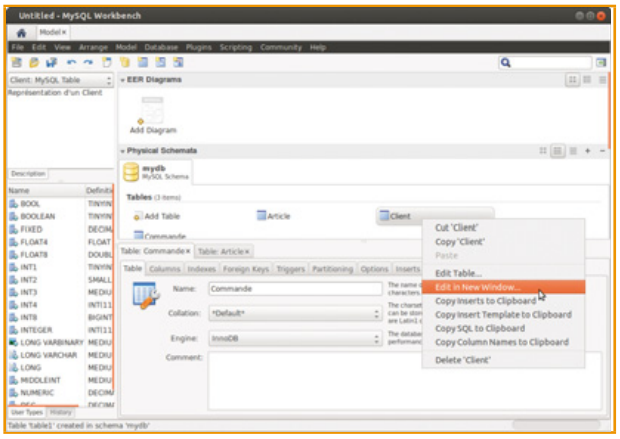


Fig. 7 : Édition de plusieurs tables simultanément en utilisant l'entrée « Edit in new Window »

1.2.2 Créer un schéma

Une fois les tables créées, nous pouvons générer le schéma en double-cliquant sur le bouton ***Add Diagram*** (zone 3). Un nouvel onglet (***EER Diagram***) est alors ouvert. Sur cet onglet, on retrouve principalement deux zones, comme le montre la figure 8 : la zone 1 qui contient les objets créés précédemment (ici nos trois tables) et la zone 2 correspondant à l'espace de création du schéma.

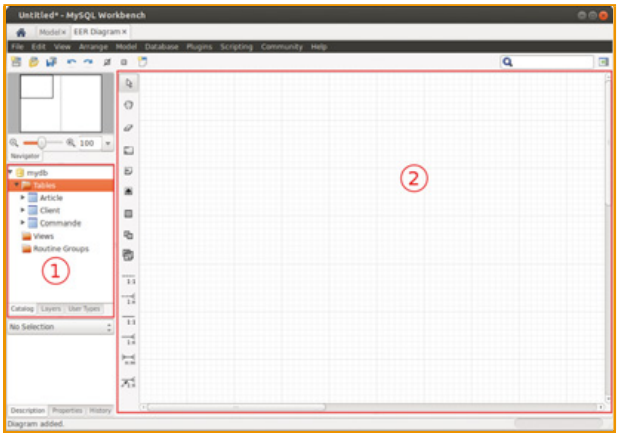


Fig. 8 : Onglet de création d'un schéma de l'outil Data Modeling

En cliquant sur les tables, vous pourrez les disposer sur le graphe par glisser-déposer. Ceci permet d'obtenir un joli schéma qui pourra être imprimé et facilitera grandement le travail de développement (et plus tard, de maintenance de la base). À ce propos, pensez au logiciel PosteRazor **[1] [2]**, qui permet d'imprimer une image sous forme de poster constitué d'autant de feuilles A4 que vous le souhaitez.

Une fois vos tables disposées, vous pouvez les organiser par groupes à l'aide des calques (*layers*). Il s'agit de zones rectangulaires englobant des tables sélectionnées généralement pour leurs interactions. On peut lire ainsi

beaucoup plus simplement le schéma et retrouver les tables à partir de leur « groupe », comme le montre la figure 9. En sélectionnant un objet (table ou calque) et en cliquant sur l'onglet ***Properties*** en bas de la fenêtre, vous pourrez modifier son apparence (taille, couleur, etc.).

Il est également possible d'ajouter des tables directement de manière graphique. La description de la table se fera ensuite dans le même onglet que précédemment, mais cette fois il sera ouvert par dessus le schéma.

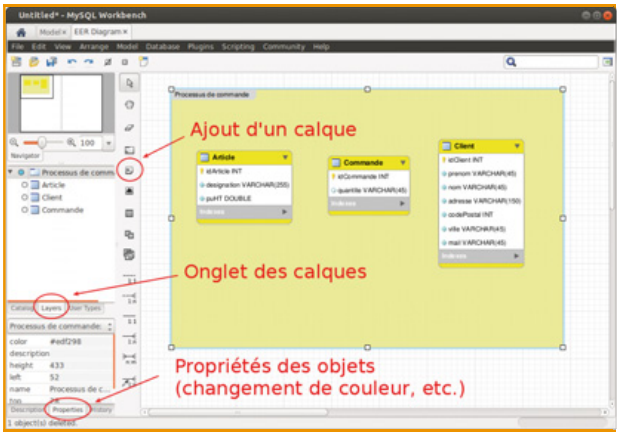


Fig. 9 : Regroupement de tables à l'intérieur d'un calque

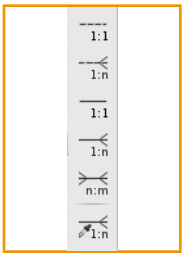


Fig. 10 : Icônes des associations disponibles dans la barre d'outils

Mais il y a mieux... Nous n'avions pas indiqué les relations entre les tables. Or, un client peut commander un article et le MLD nous indiquait que la table **`Commande`** devait contenir deux clés étrangères pointant vers les tables **`Client`** et **`Article`**. Pour cela, nous n'avons qu'à utiliser les icônes d'associations (Fig. 10), puis à cliquer sur chacune des tables participant à la relation. L'association sera alors automatiquement créée et les modifications structurelles du schéma seront appliquées (ajout d'un champ, d'une table, etc.). Nous obtenons ainsi le schéma de la figure 11, où deux clés étrangères ont été ajoutées à la table **`Commande`**. Si vous souhaitez changer leur nom, il faudra éditer la table et choisir l'onglet ***Foreign Keys*** (Fig. 12). Notez à droite de l'onglet la possibilité de renseigner les actions à effectuer en cas de modification ou de suppression des enregistrements auxquels les clés étrangères font référence.

Maintenant que le schéma est créé, nous pouvons passer à la création de la base.

1.2.3 Créer une base

Le menu ***Database > Forward Engineer*** permet de générer le code SQL, de se connecter à un serveur MySQL ou MariaDB (il vous faudra préciser vos paramètres de connexion) et

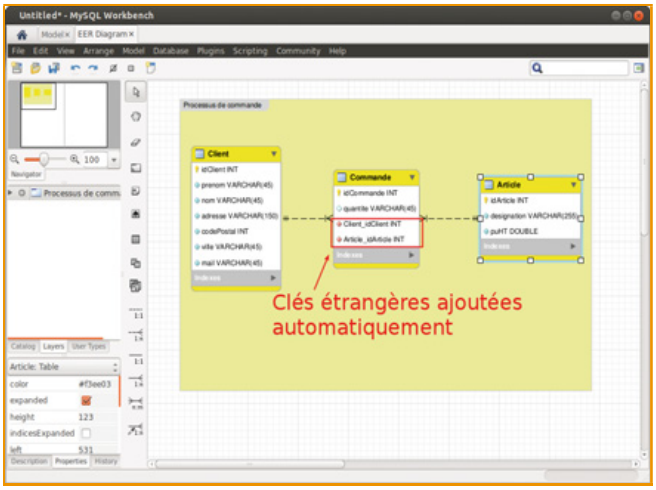


Fig. 11 : Schéma avec ajout de relations entre les tables

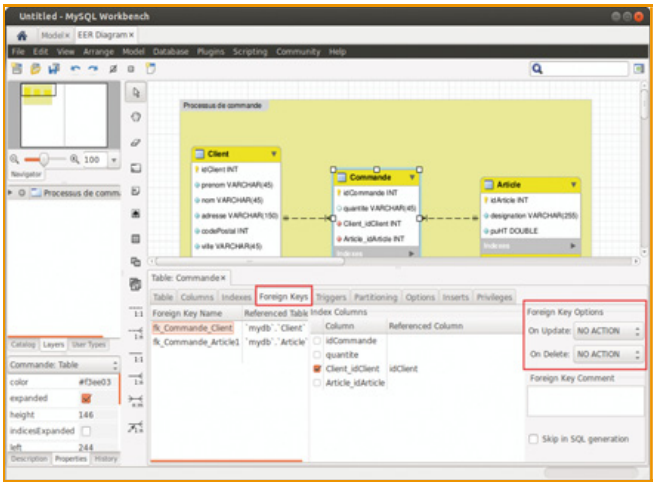


Fig. 12 : Modification des clés étrangères d'une table

de créer la base. De nombreuses options sont disponibles permettant, notamment, de sélectionner les objets à exporter dans la base. Pour une utilisation sur de petits projets, les options par défaut conviendront parfaitement.

Voici un extrait du code généré pour notre schéma :

```
01: CREATE SCHEMA IF NOT EXISTS `maBase` DEFAULT CHARACTER SET utf8;
02: USE `maBase` ;
03:
04: -- -----------------------------------------------------
05: -- Table `maBase`.`Client`
06: -- -----------------------------------------------------
07: CREATE  TABLE IF NOT EXISTS `maBase`.`Client` (
08:   `idClient` INT NOT NULL AUTO_INCREMENT ,
09:   `prenom` VARCHAR(45) NOT NULL ,
10:   `nom` VARCHAR(45) NOT NULL ,
11:   `adresse` VARCHAR(150) NOT NULL ,
12:   `codePostal` INT NOT NULL ,
13:   `ville` VARCHAR(45) NOT NULL ,
14:   `mail` VARCHAR(45) NOT NULL COMMENT 'Adresse mail du
client\npour le suivi de la\ncommande' ,
15:   PRIMARY KEY (`idClient`) )
```

```
16: ENGINE = InnoDB
17: COMMENT = 'Représentation d\'un Client';
18:
19:
20: -- -----------------------------------------------------
21: -- Table `maBase`.`Article`
22: -- -----------------------------------------------------
23: CREATE  TABLE IF NOT EXISTS `maBase`.`Article` (
24:   `idArticle` INT NOT NULL AUTO_INCREMENT ,
25:   `designation` VARCHAR(255) NOT NULL ,
26:   `puHT` DOUBLE NOT NULL ,
27:   PRIMARY KEY (`idArticle`) )
28: ENGINE = InnoDB;
29:
30:
31: -- -----------------------------------------------------
32: -- Table `maBase`.`Commande`
33: -- -----------------------------------------------------
34: CREATE  TABLE IF NOT EXISTS `maBase`.`Commande` (
35:   `idCommande` INT NOT NULL AUTO_INCREMENT ,
36:   `quantite` VARCHAR(45) NULL ,
37:   `Client_idClient` INT NOT NULL ,
38:   `Article_idArticle` INT NOT NULL ,
39:   PRIMARY KEY (`idCommande`) ,
40:   INDEX `fk_Commande_Client` (`Client_idClient` ASC) ,
41:   INDEX `fk_Commande_Article1` (`Article_idArticle` ASC) ,
42:   CONSTRAINT `fk_Commande_Client`
43:     FOREIGN KEY (`Client_idClient` )
44:     REFERENCES `maBase`.`Client` (`idClient` )
45:     ON DELETE NO ACTION
46:     ON UPDATE NO ACTION,
47:   CONSTRAINT `fk_Commande_Article1`
48:     FOREIGN KEY (`Article_idArticle` )
49:     REFERENCES `maBase`.`Article` (`idArticle` )
50:     ON DELETE NO ACTION
51:     ON UPDATE NO ACTION)
52: ENGINE = InnoDB;
```

On distingue bien la création et la définition de la base à utiliser dans les lignes 1 et 2 puis, la définition des trois tables : **`Client`** dans les lignes 4 à 17, **`Article`** dans les lignes 20 à 28 et **`Commande`** dans les lignes 31 à 52. Il s'agit d'un code SQL que vous pouvez exécuter directement sur votre serveur de base de données si vous avez créé une connexion dans l'outil SQL Development. Sur la figure 13, nous utilisons la connexion créée en début d'article pour exécuter le code SQL et créer notre base.

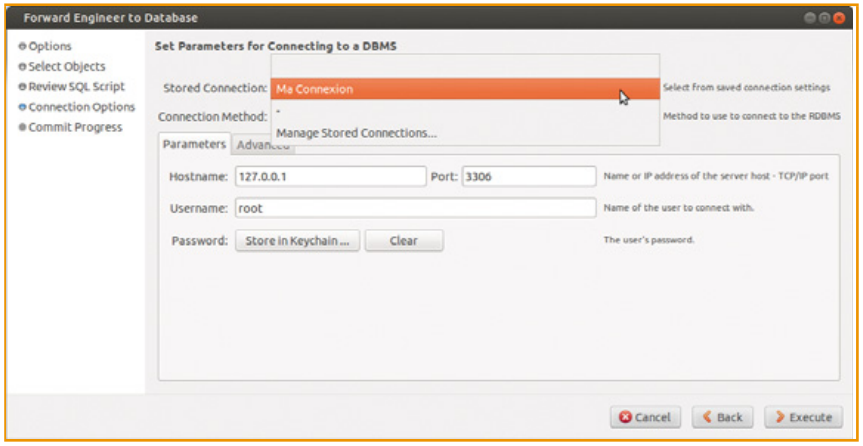


Fig. 13 : Utilisation d'une connexion existante pour exécuter le code de génération de la base

1.2.4 Pour aller plus loin : le reverse engineering

Dans le cas d'un projet existant et sur lequel vous ne disposez d'aucune documentation sur le schéma de base de données, la seule solution consiste à effectuer du reverse engineering : à partir de la définition des tables, MySQL Workbench va tenter de retrouver le schéma. Attention : comme tout reverse engineering, la réussite totale de l'opération n'est pas garantie si le schéma est complexe et vous aurez au moins la moitié du travail à faire manuellement !

La réalisation de cette opération est très simple. Il faut se rendre dans le menu ***Database > Reverse Engineer*** qui ouvrira une boîte de dialogue permettant de se connecter au serveur de base de données (la même boîte que précédemment en figure 13). Les différentes bases en seront extraites et il suffira de sélectionner celle dont on désire obtenir le schéma. La figure 14, page suivante, montre le schéma d'un petit projet de test utilisant le framework Python Django. Comme vous le voyez, même avec peu de tables, la première étape va être de « démêler » tout ça...

1.3 Server Administration

Tout comme avec l'outil SQL Developement où il fallait créer une connexion pour démarrer, il va falloir ici créer une instance d'un serveur en cliquant sur le bouton ***New Server Instance***. Il n'y a qu'à suivre les différents écrans pour entrer les différents paramètres de connexion et créer la nouvelle instance (il vous sera également demandé le type d'installation de MySQL sur le serveur : paquetage (quelle distribution ?), depuis les sources, etc.).Une fois l'instance créée, elle apparaîtra dans la liste de l'interface générale de MySQL Workbench et en cliquant dessus vous ouvrirez un nouvel onglet ***Server Administration*** relatif au serveur configuré. Vous aurez ainsi une vue de contrôle du système (Fig. 15, page suivante), la possibilité d'ajouter, modifier ou supprimer des utilisateurs (menu ***Users and Privileges***), etc.

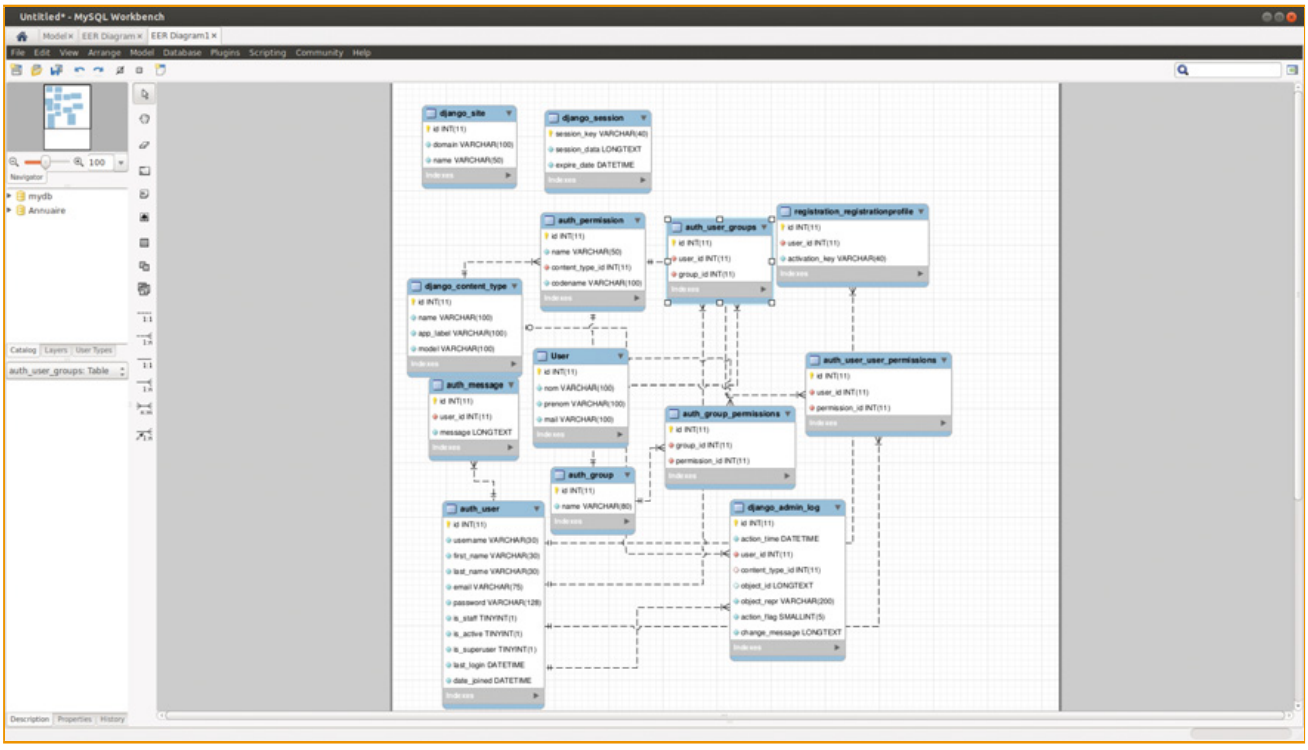


Fig. 14 : Obtention d'un schéma par reverse engineering

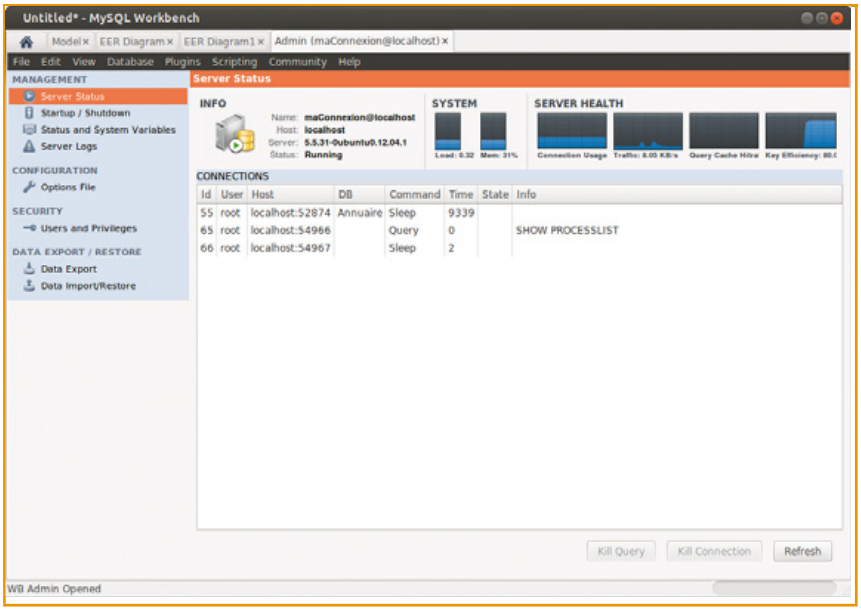


Fig. 15 : Surveillance d'un serveur de base de données MySQL à l'aide de l'outil Server Administration

2 SQL Power Architect

Après avoir vu un outil dédié à MySQL/MariaDB, voici un outil pour PostgreSQL (en fait, il fonctionne aussi pour MySQL, MariaDB et beaucoup d'autres systèmes de gestion de bases de données). SQL Power Architect est décliné en plusieurs versions, dont une version communautaire distribuée sous licence GPL version 3. Il n'existe pas de paquetage pour ce logiciel et il faut donc télécharger l'archive compressée sur http://www.sqlpower.ca/page/architect_download_os. La première étape sera bien sûr de décompresser le fichier obtenu. Je placerai le répertoire résultant dans **/opt** :

```
sudo tar -zxvf SQL-Power-Architect-generic-jdbc-1.0.6.tar.gz -C /opt
```

Le répertoire **architect-1.0.6** issu de la décompression contient un fichier **architect.jar** que l'on lance à l'aide de la commande :

```
java -jar architect.jar
```

À la première exécution, il vous sera demandé de confirmer la création d'un fichier de configuration **pl.ini** dans votre répertoire personnel. Puis, l'interface apparaîtra... vide : il y a une barre de menu et une barre d'outils générale en haut de la fenêtre et une deuxième barre d'outils permettant de créer un schéma sur la droite (Fig. 16).

Pour se connecter à une base de données, il faut créer une nouvelle connexion en sélectionnant dans le menu ***Connections > Add Source Connection > New Connection***. Une boîte de dialogue similaire à celle présentée en figure 17 apparaîtra et vous devrez compléter les divers champs permettant la connexion à la base . Sélectionnez bien le type de votre base de données : comme ici nous voulons nous connecter à une base PostgreSQL, c'est bien sûr ce SGBD qu'il faudra choisir. Le bouton ***Test Connection*** permet de vérifier que la connexion peut s'effectuer sans problème avant de valider. Le nom de votre nouvelle connexion apparaîtra dans la colonne de gauche de la fenêtre principale, sous ***PlayPen Database***.

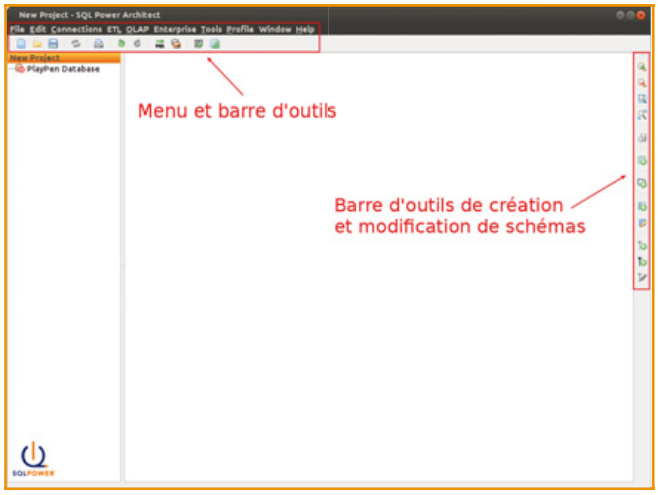


Fig. 16 : Page de démarrage de SQL Power Architect Community Edition

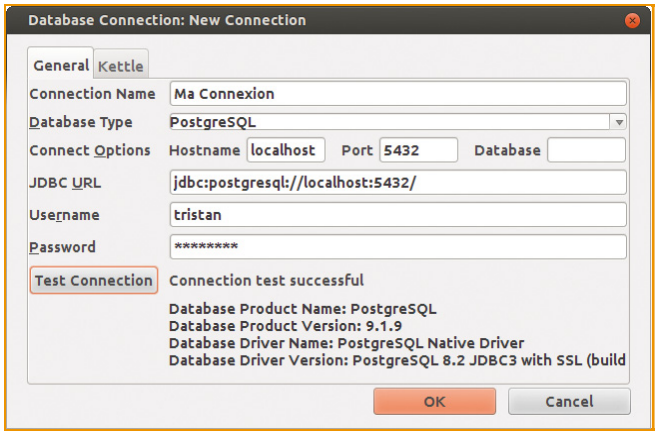


Fig. 17 : Création d'une nouvelle connexion dans SQL Power Architect

2.1 Créer un schéma

Pour créer un schéma, il faut utiliser la barre d'outils de gauche. En reprenant notre exemple **Client – commander – Article**, nous pouvons créer la table `Client` en cliquant sur le bouton ***New Table*** (le sixième en partant du haut) et en complétant la boîte de dialogue qui s'affiche alors (Fig. 18). L'ajout de champs se fera directement sur le schéma, en effectuant un clic droit et en sélectionnant ***New Column***. Une nouvelle boîte de dialogue apparaîtra, permettant de saisir toutes les informations relatives au nouveau champ (Fig. 19).

La présentation est différente, mais l'on reste sur la même philosophie que dans MySQL Workbench. On peut donc rapidement créer les trois tables et, toujours comme avec l'outil MySQL Workbench, en ajoutant les relations sur le schéma (deuxième bouton de la barre d'outils en partant du bas), la table `Commande` sera automatiquement modifiée pour y incorporer les deux clés étrangères. La figure 20, page suivante, montre le schéma obtenu.

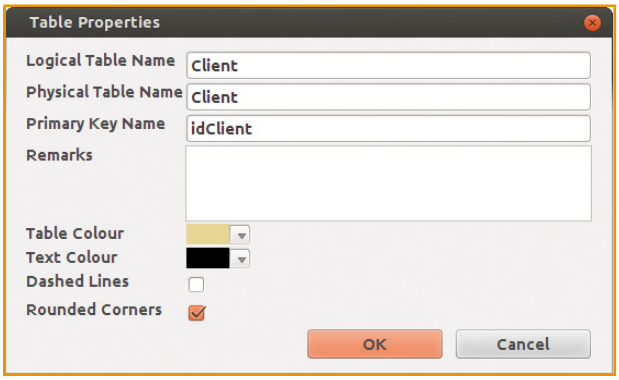


Fig. 18 : Boîte de dialogue de création d'une table

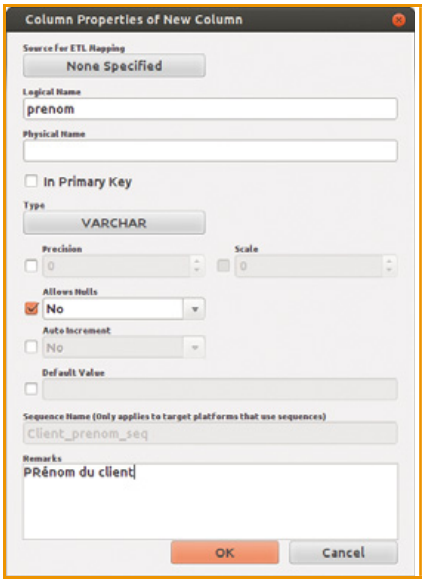


Fig. 19 : Boîte de dialogue d'ajout d'un champ à une table

2.2 Créer une base

En cliquant dans le menu ***Tools > Forward Engineer***, vous pourrez choisir une connexion (donc un serveur PostgreSQL) sur laquelle exécuter le code SQL généré à partir du schéma. Vous pourrez constater que le code proposé est différent et moins bien présenté que celui généré par MySQL Workbench pour une base MySQL. En voici un extrait :

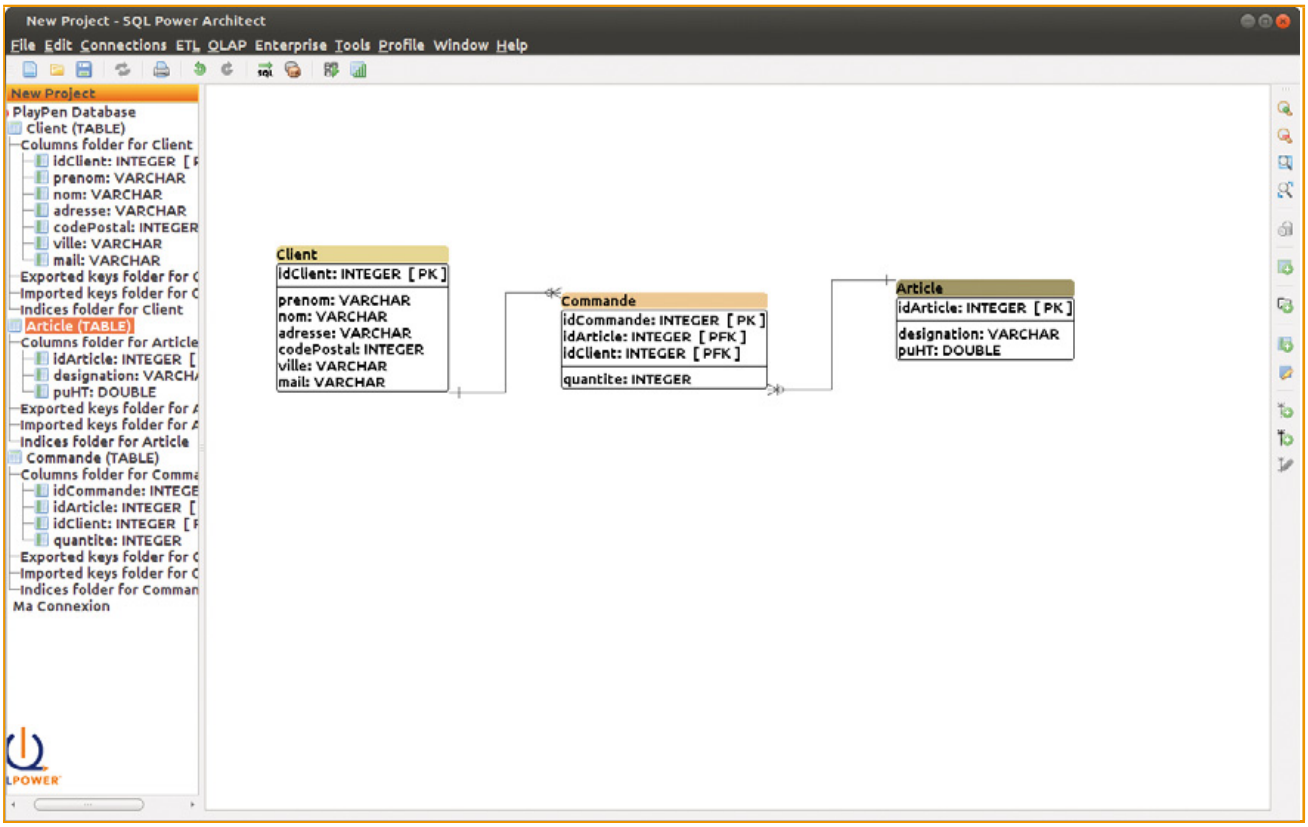


Fig. 20 : Schéma de la base Client – Commande – Article dans SQL Power Architect

```
01: CREATE SEQUENCE public.article_idarticle_seq;
02:
03: CREATE TABLE public.Article (
04:                 idArticle INTEGER NOT NULL DEFAULT nextval('public.article_idarticle_seq'),
05:                 designation VARCHAR NOT NULL,
06:                 puHT DOUBLE PRECISION NOT NULL,
07:                 CONSTRAINT idarticle PRIMARY KEY (idArticle)
08: );
09:
10:
11: ALTER SEQUENCE public.article_idarticle_seq OWNED BY public.Article.idArticle;
```

Il n'y a plus qu'à cliquer sur le bouton ***Execute*** pour que la base soit créée sur le serveur.

2.3 Pour aller plus loin : le reverse engineering

Le reverse engineering est également disponible avec SQL Power Architect : ouvrez une base dans la colonne de gauche (à partir d'une connexion créée) et effectuez un glisser-déposer des tables que vous souhaitez utiliser vers la zone de schéma. Le bouton ***Automatic Layout*** de la barre d'outils du haut (avant-dernier bouton) calculera automatiquement la position des objets du schéma pour que celui-ci soit lisible.

Conclusion

Nous avons pu voir que les fonctionnalités de MySQL Workbench et SQL Power Architect étaient très proches. MySQL Workbench a un design et une ergonomie plus travaillés et propose, outre la création de schémas et le reverse engineering, des outils de gestion et de supervision du SGBD. De son côté, SQL Power Architect propose moins d'outils, mais permet de travailler avec de nombreux types de SGBD.

Si vous travaillez sur une base MySQL ou MariaDB, je pense que le choix sera vite fait et que vous vous orienterez vers MySQL Workbench. Malheureusement, pour du PostgreSQL, il faudra passer à SQL Power Architect qui, bien que fonctionnant très bien et produisant au final le même résultat que MySQL Workbench, est bien moins souple à utiliser... et bien moins joli ! ■

Bibliographie

[1] COLOMBO (Tristan), « Créez vos posters avec PosteRazor ! », Linux Pratique n° 55, Septembre 2009, p. 50 et 51.

[2] Site officiel de PosteRazor : http://posterazor.sourceforge.net/

Complétez votre collection d'anciens numéros !

ed-diamond.com

numerique.ed-diamond.com

LE LANGAGE SQL

par Tristan Colombo

SQL est un langage normalisé, donc indépendant des systèmes de gestion de bases de données. Il permet d'interagir avec les bases sous la forme de requêtes qui seront, à peu de différences près, écrites de la même manière quel que soit le SGBD choisi. Ce langage représente donc le cœur de la communication avec les SGBD et il est important de bien le maîtriser.

Le langage SQL (pour *Structured Query Language*), auparavant SEQUEL (pour *Structured English as QUEry Language*) et développé initialement par IBM, a été normalisé en 1987 (SQL1). Depuis, 5 autres normes ont été définies : SQL2 ou SQL-92 en 1992 et SQL3 ou SQL-99 en 1999, avec notamment l'ajout de concepts objets, puis SQL:2003, SQL:2008 et SQL:2011. En général, les SGBD « classiques » tels que MySQL/MariaDB, PostgreSQL, etc., supportent a minima la norme SQL2 et implémentent certaines parties des normes supérieures **[1][2]**.

Toutes les fonctions du langage normatif SQL sont assurées par des commandes (ou ordres) classées en 4 groupes :

- DDL (*Data Definition Language*) : définition des données avec les commandes **CREATE**, **ALTER**, **DROP** et **RENAME** ;
- DML (*Data Manipulation Language*) : manipulation des données avec les commandes **INSERT**, **DELETE**, **UPDATE**;
- DQL (*Data Query Language*) : interrogation des données avec la commande la plus utilisée, **SELECT** ;
- DCL (*Data Control Language*) : contrôle des données avec les commandes **GRANT**, **REVOQUE**, **COMMIT**, **ROLLBACK**.

Dans cet article, j'utiliserai une base MySQL depuis le terminal. La connexion au serveur se fait en spécifiant un nom d'utilisateur, puis on obtient un entête et un prompt **mysql>** :

```
login@server:~$ mysql --user=root -p
Enter password:
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 48
Server version: 5.5.31-0ubuntu0.12.04.1 (Ubuntu)

Copyright (c) 2000, 2013, Oracle and/or its affiliates. All rights reserved.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

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

mysql>
```

Des informations importantes vous sont données dans cet entête. Ces dernières sont écrites en jaune :

- Toutes les commandes se terminent par un point-virgule ou **\g** ;
- L'aide est accessible par **help;** (toujours le point-virgule à la fin de la commande) ou **\h** ;
- **\c** permet de ne pas prendre en compte le contenu précédent sur la ligne sans avoir à tout effacer.

Comme vous pouvez le voir, le prompt est très rudimentaire : il ne donne aucune information pertinente si ce n'est que nous travaillons sur un serveur MySQL. Il est possible de le configurer en utilisant l'option **--prompt** au lancement de **mysql** ou en utilisant la commande **\R** dans **mysql**. Dans les deux cas, il faudra fournir une chaîne de caractères formatée indiquant les informations à afficher. Les séquences spéciales utilisables dans ce contexte sont données par un appel à **man mysql**. Vous en trouverez quelques-unes dans le tableau suivant :

Séquence	Affichage
\u	Nom de l'utilisateur
\h	Nom du serveur
\U	Équivalent de **\u@\h**
\v	Version du serveur
\d	Base de données en cours d'utilisation
\D	Date complète
\c	Compteur d'instructions
_	Un espace

Pour obtenir un prompt indiquant avec quel utilisateur nous travaillons, sur quel serveur et sur quelle base, nous pouvons exécuter :

```
mysql> \R \U [\d]>
PROMPT set to '\U [\d]> '
root@localhost [(none)]>
```

Comme nous ne travaillons encore sur aucune base de données, nous obtenons l'affichage **(none)**. Nous avons bien modifié le prompt, mais si nous quittons MySQL et que nous le relançons, nous verrons que la modification n'est pas persistante... Et nous n'allons pas modifier le prompt à chaque nouvelle connexion ! Éditez donc le fichier **/etc/mysql/my.cnf** en tant qu'administrateur et ajoutez dans la section **[mysql]** la ligne suivante :

```
prompt   = \U [\d]>\_
```

Désormais, chaque fois que vous lancerez la commande **mysql**, vous obtiendrez l'affichage du prompt que vous avez personnalisé.

Dans la suite, je vais présenter le langage SQL en suivant les 4 groupes DDL, DML, DQL et DCL. Nous commencerons donc par créer une base de données et des tables, puis nous insérerons des données, ensuite nous récupérerons des données dans la base et enfin, nous aborderons les commandes de contrôle.

Écrire du code SQL : majuscules ou minuscules ?

Le langage SQL n'est pas sensible à la casse (différence entre les majuscules et les minuscules) pour les commandes, alors qu'il l'est pour les identifiants. Vous pourrez trouver du code SQL où les commandes sont exclusivement écrites en majuscules et d'autres codes où elles ne le seront qu'en minuscules. Quelle est la bonne solution ? De manière générale, on choisit d'écrire le code en majuscules lorsqu'il n'y a pas de coloration syntaxique pour faire ressortir les instructions. Dans les autres cas, on écrit en minuscules. Dans le cadre de cet article, j'ai opté pour l'écriture en majuscules pour plus de clarté, mais sachez donc que si vous tapez le code en minuscules il fonctionnera aussi.

1 DDL ou définition des données

Pour créer des objets (bases, tables, vues, etc.), on utilise la commande **CREATE** associée au nom de l'objet que l'on désire créer. Avant de pouvoir créer quoi que ce soit, nous devons disposer d'une base ou « schéma ».

1.1 CREATE

1.1.1 Création d'une base de données (ou schéma)

Nous allons une nouvelle fois utiliser l'exemple du client qui commande des articles. Notre base s'appellera **siteMarchand** (attention, SQL est sensible à la casse pour les identifiants). Pour la créer, nous utiliserons la commande **CREATE SCHEMA** ou **CREATE DATABASE** qui est équivalente. Pour bien voir que notre base a été créée, nous allons la sélectionner comme base de données courante à l'aide de la commande **USE** :

```
root@localhost [(none)]> CREATE DATABASE siteMarchand;
Query OK, 1 row affected (0.00 sec)

root@localhost [(none)]> USE siteMarchand;
Database changed
root@localhost [siteMarchand]>
```

Nous voyons que le prompt a bien été modifié pour nous indiquer sur quelle base de données nous travaillons maintenant. Encore une fois, n'oubliez pas le point-virgule en fin de ligne si vous souhaitez que vos commandes puissent être exécutées...

Pour afficher la liste des bases de données disponibles, on peut également utiliser la commande **SHOW** (cette commande permet aussi d'afficher d'autres objets, comme nous le verrons par la suite) :

```
root@localhost [siteMarchand]> SHOW DATABASES;
+--------------------+
| Database           |
+--------------------+
| information_schema |
| Annuaire           |
| mysql              |
| performance_schema |
| phpmyadmin         |
| siteMarchand       |
| test               |
+--------------------+
7 rows in set (0.00 sec)
```

Les résultats sont présentés sous la forme d'une table en mode texte et nous retrouvons notre base **siteMarchand**.

1.1.2 Création d'une table

Une table sera créée à l'aide de la commande **CREATE TABLE** suivie du nom de la table et de la liste de ses colonnes dont le type devra être spécifié. Des commandes spécifiques permettent d'indiquer :

- une valeur par défaut : **DEFAULT** ;
- une valeur non nulle : **NOT NULL** ;
- un commentaire : **COMMENT** ;

- une valeur unique : **UNIQUE** ;
- une valeur s'incrémentant automatiquement : **AUTO_INCREMENT** ;
- une clé primaire : **PRIMARY KEY** ;
- une clé étrangère : **FOREIGN KEY**.

Pour créer la table **Client**, il faudra exécuter :

```
root@localhost [siteMarchand]> CREATE TABLE Client (
    -> idClient INT NOT NULL AUTO_INCREMENT,
    -> prenom VARCHAR(45) NOT NULL,
    -> nom VARCHAR(45) NOT NULL,
    -> adresse VARCHAR(150) NOT NULL,
    -> codePostal INT NOT NULL,
    -> ville VARCHAR(45) NOT NULL,
    -> mail VARCHAR(100) NOT NULL,
    -> PRIMARY KEY (idClient))
    -> COMMENT = 'Table definissant un client';
Query OK, 0 rows affected (0.10 sec)
```

La commande **SHOW TABLES** permet de vérifier que nous avons bien créé la table **Client** dans la base de données active :

```
root@localhost [siteMarchand]> SHOW TABLES;
+----------------------+
| Tables_in_siteMarchand |
+----------------------+
| Client               |
+----------------------+
1 row in set (0.00 sec)
```

Dans le cas où vous n'auriez activé aucune base, vous pouvez créer la table **Client** dans la base **siteMarchand** en préfixant le nom de la table par le nom de la base :

```
root@localhost [(none)]> CREATE TABLE siteMarchand.Client (...);
```

Pour afficher dans le détail les informations relatives à une base (et le commentaire que nous avions ajouté), il faut exécuter :

```
root@localhost [siteMarchand]> SHOW TABLE STATUS LIKE 'Client';
```

La table **Article** est créée en utilisant les mêmes commandes. Pour la table **Commande** par contre, il va nous falloir définir des clés étrangères :

```
root@localhost [siteMarchand]> CREATE TABLE Commande (
    -> idCommande INT NOT NULL AUTO_INCREMENT,
    -> fk_idArticle INT NOT NULL,
    -> fk_idClient INT NOT NULL,
    -> PRIMARY KEY (idCommande),
    -> FOREIGN KEY (fk_idArticle) REFERENCES Article(idArticle),
    -> FOREIGN KEY (fk_idClient) REFERENCES Client(idClient));
Query OK, 0 rows affected (0.18 sec)
```

En cas d'erreur lors de la création de votre table et des clés étrangères (mauvais nom, mauvais type, etc.), vous obtiendrez une erreur **1005** :

```
" 1005 (ER_CANT_CREATE_TABLE)
```

Impossible de créer la table. Si le message d'erreur fait référence à une erreur de code errno 150, la création de la table a échoué à cause d'une contrainte de clé étrangère, qui n'est pas correctement formée. » (source : manuel de référence MySQL **[3]**).

1.1.3 Création d'une vue

Nous avions vu dans l'article sur les bases de données relationnelles qu' « *Une vue est une sorte de table virtuelle créée à partir d'une requête.* »... Il est temps de voir la requête. Nous allons être obligés d'utiliser ici la commande **SELECT** qui fait partie du groupe DQL (interrogation des données). Comme nous traiterons cette commande par ailleurs, nous considérerons qu'elle ne peut que sélectionner les valeurs d'un certain nombre de colonnes d'une table. Ainsi, pour sélectionner uniquement les colonnes **nom** et **prenom** de la table **Client**, il faut écrire :

```
SELECT nom, prenom FROM Client;
```

Si nous utilisons fréquemment cette requête pour ne connaître que les noms et prénoms des clients, alors il est préférable de créer une vue :

```
root@localhost [siteMarchand]> CREATE VIEW Summary AS
    -> SELECT prenom, nom FROM Client;
Query OK, 0 rows affected (0.05 sec)
```

La vue **Summary** est maintenant considérée comme une table classique, à la différence qu'on ne pourra pas y insérer de données, mettre à jour ou supprimer un enregistrement. La commande **SHOW TABLES** affichera d'ailleurs notre vue :

```
root@localhost [siteMarchand]> SHOW TABLES;
+----------------------+
| Tables_in_siteMarchand |
+----------------------+
| Article              |
| Client               |
| Commande             |
| Summary              |
+----------------------+
4 rows in set (0.00 sec)
```

1.1.4 Création d'un index

Un index permet d'éviter de parcourir les informations d'une table du premier enregistrement au dernier lorsque l'on y recherche des données. Retrouver des données dans une table sans index c'est comme rechercher dans un dictionnaire sans index : s'il y a peu d'informations, on doit pouvoir retrouver les données dans un temps à peu près similaire (une seule page)... Mais dès que l'on augmente la quantité de données, l'index permet d'accélérer grandement les recherches. Sur une table contenant 1000 enregistrements, la recherche sera 100 fois plus rapide avec un index !

idClient	prenom	nom	adresse	codePostal	ville	mail
1	Jim	Shannon	Hadrosaurus street	2522	Terra Nova	js@tn.com
2	Laura	Reily	Stegosaurus street	2523	Terra Nova	lr@tn.com
3	Alicia	Washington	Pterausaurus street	3005	Terra Nova	aw@tn.com
4	Malcolm	Wallace	Brachiosaurus road	2684	Terra Nova	mw@tn.com
5	Nathaniel	Taylor	Acceraptor avenue	3004	Terra Nova	nt@tn.com
6	Skye	Tate	Velociraptor road	2233	Terra Nova	st@tn.com
7	Tom	Boylan	Spinosaurus street	1245	Terra Nova	tb@tn.com
8	Mark	Reynolds	Allosaurus avenue	2056	Terra Nova	mr@tn.com

La clé primaire est un index, mais l'on n'effectue pas forcément des recherches à partir de la clé primaire. Supposons que l'on recherche un client par son nom et que l'on dispose de la table **Client** suivante (nous verrons plus loin comment insérer des données dans une table) : voir tableau ci-dessus.

Si l'on recherche « Reynolds », il faudra parcourir 8 enregistrements. Pour simplifier, imaginons que la création d'un index va créer un arbre binaire balancé (aussi connu sous le nom de *B-tree* ou arbre B). Cet exemple n'est pas le reflet de la réalité, car un index n'est pas un arbre binaire (un nœud peut posséder plus de 2 fils). La figure 1 montre l'arbre créé à partir des données de la table en utilisant le champ **nom** pour index.

Si l'on recherche toujours « Reynolds », cette fois l'opération ne devra plus parcourir que 3 niveaux dans l'arbre pour retrouver l'identifiant de colonne (dans cet exemple, la clé primaire **idClient**). L'enregistrement est retrouvé 2,6 fois plus vite. Même en disposant de très peu de données, l'accélération est loin d'être négligeable !

D'un point de vue syntaxique, l'ajout d'un index sur une table se fait a posteriori grâce à la commande **CREATE INDEX** :

```
root@localhost [siteMarchand]> CREATE INDEX idx_Client_nom
    -> USING BTREE
    -> ON Client (nom);
Query OK, 0 rows affected (0.17 sec)
Records: 0  Duplicates: 0  Warnings: 0
```

Nous avons spécifié ici que notre index utilisait un **BTREE**, car MySQL peut aussi travailler avec des **HASH** (table de hachage).

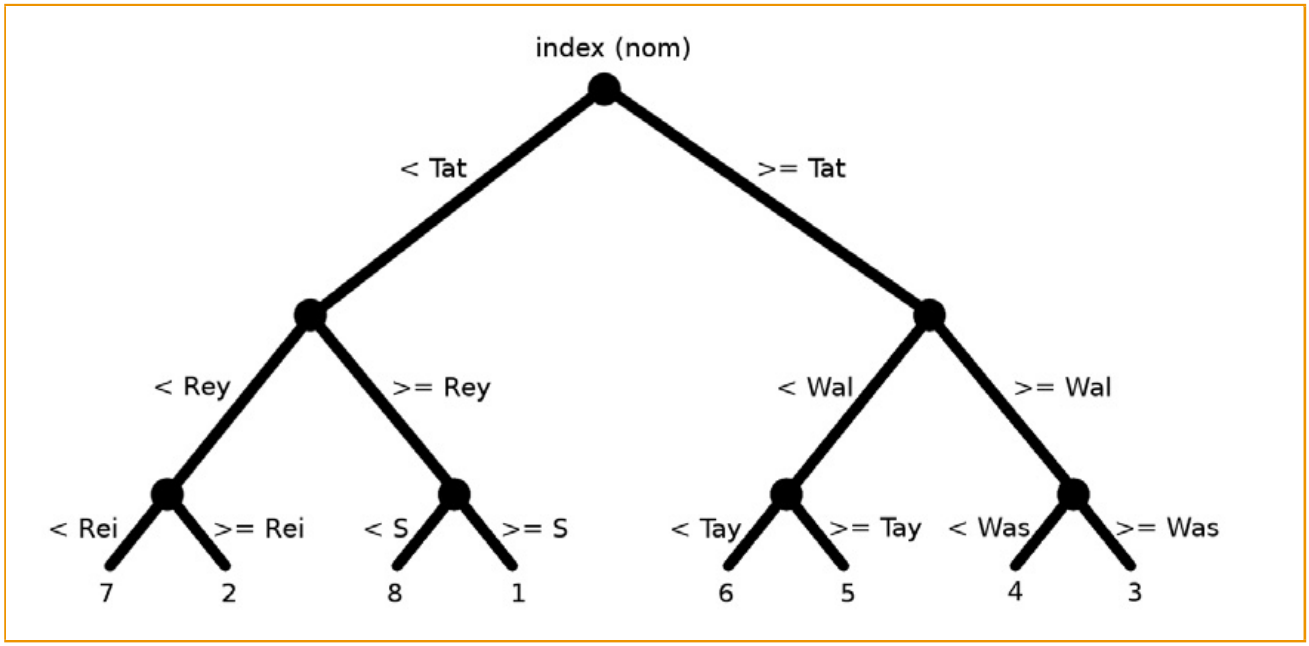


Fig. 1 : Arbre binaire balancé correspondant aux données de la table Client avec pour index le champ **nom**

1.2 RENAME

Vous pouvez renommer des tables à l'aide de la commande **RENAME**. Attention, contrairement au **CREATE**, **RENAME** ne fonctionne qu'avec des tables :

```
root@localhost [siteMarchand]> RENAME TABLE Article TO Art;
Query OK, 0 rows affected (0.05 sec)
```

On peut également faire des choses plus exotiques :

```
root@localhost [siteMarchand]> RENAME TABLE siteMarchand.Article
TO autreBase.Art;
Query OK, 0 rows affected (0.05 sec)
```

Ici, la table **Article** de la base **siteMarchand** a été renommée en **Art** et déplacée dans la base **autreBase**.

Si vous souhaitez renommer une base de données, il vous faudra créer une nouvelle base, déplacer toutes les tables de l'ancienne base vers la nouvelle, et supprimer l'ancienne base. C'est l'exemple que nous allons mettre en pratique dans la partie suivante.

1.3 DROP

Pour supprimer un objet, il faut utiliser la commande **DROP** suivie du type d'objet que l'on souhaite supprimer (**TABLE**, **VIEW**, **DATABASE**, etc.). La suppression de la vue **Summary** se fera par exemple par :

```
root@localhost [siteMarchand]> DROP VIEW Summary;
Query OK, 0 rows affected (0.00 sec)
```

Maintenant, si nous voulons appliquer le processus complet permettant de renommer une base de données, il faudra exécuter :

```
root@localhost [(none)]> CREATE DATABASE autreBase;
Query OK, 1 row affected (0.02 sec)

root@localhost [(none)]> RENAME TABLE siteMarchand.Article TO
autreBase.Article;
Query OK, 0 rows affected (0.05 sec)

root@localhost [(none)]> RENAME TABLE siteMarchand.Commande TO
autreBase.Commande;
Query OK, 0 rows affected (0.05 sec)

root@localhost [(none)]> RENAME TABLE siteMarchand.Client TO
autreBase.Client;
Query OK, 0 rows affected (0.04 sec)

root@localhost [(none)]> DROP DATABASE siteMarchand;
Query OK, 0 rows affected (0.00 sec)
```

1.4 ALTER

La commande **ALTER** permet d'effectuer des modifications structurelles sur un objet. Dans une table, nous pourrons par exemple ajouter une colonne, renommer une colonne, modifier le type d'une colonne, supprimer une colonne, etc. Commençons par ajouter une colonne « age » dans la table **Client** :

```
root@localhost [siteMarchand]> ALTER TABLE Client ADD age INT NOT NULL
AFTER prenom;
Query OK, 0 rows affected (0.23 sec)
Records: 0  Duplicates: 0  Warnings: 0
```

Nous ajoutons le champ « age » après le champ « prenom » sous la forme d'un entier non nul. Pour vérifier que la colonne a bien été ajoutée, nous pouvons utiliser la commande **SHOW** suivie de **COLUMNS** :

```
root@localhost [siteMarchand]> SHOW COLUMNS FROM Client;
+-----------+-------------+------+-----+---------+----------------+
| Field     | Type        | Null | Key | Default | Extra          |
+-----------+-------------+------+-----+---------+----------------+
| idClient  | int(11)     | NO   | PRI | NULL    | auto_increment |
| prenom    | varchar(45) | NO   |     | NULL    |                |
| age       | int(11)     | NO   |     | NULL    |                |
| nom       | varchar(45) | NO   | MUL | NULL    |                |
| adresse   | varchar(150)| NO   |     | NULL    |                |
| codePostal| int(11)     | NO   |     | NULL    |                |
| ville     | varchar(45) | NO   |     | NULL    |                |
| mail      | varchar(100)| NO   |     | NULL    |                |
+-----------+-------------+------+-----+---------+----------------+
8 rows in set (0.00 sec)
```

Nous souhaitons maintenant modifier cette nouvelle colonne. Finalement, nous souhaitons conserver la date de naissance au format **date** :

```
root@localhost [siteMarchand]> ALTER TABLE Client CHANGE age naissance
date;
Query OK, 0 rows affected (0.18 sec)
Records: 0  Duplicates: 0  Warnings: 0
```

Encore une fois, un appel à **SHOW COLUMNS** nous montrera les modifications apportées :

```
root@localhost [siteMarchand]> SHOW COLUMNS FROM Client;
+-----------+-------------+------+-----+---------+----------------+
| Field     | Type        | Null | Key | Default | Extra          |
+-----------+-------------+------+-----+---------+----------------+
...
| naissance | date        | YES  |     | NULL    |                |
...
+-----------+-------------+------+-----+---------+----------------+
8 rows in set (0.00 sec)
```

La suppression d'une colonne se fera en associant la commande **DROP COLUMN** à **ALTER TABLE** :

```
root@localhost [siteMarchand]> ALTER TABLE Client DROP COLUMN
naissance;
Query OK, 0 rows affected (0.19 sec)
Records: 0  Duplicates: 0  Warnings: 0
```

Enfin, passons à une modification un peu plus profonde en ajoutant une clé étrangère. C'est une nouvelle contrainte

qui est ajoutée à la table (mot-clé **CONSTRAINT**). Pour le reste, il ne s'agit que d'une combinaison de commandes que nous avons déjà vues :

```
root@localhost [siteMarchand]> ALTER TABLE Client ADD (
    -> fk_idArticle INT NOT NULL,
    -> CONSTRAINT fk_idArticle FOREIGN KEY (fk_idArticle) REFERENCES
Article(idArticle));
Query OK, 0 rows affected (0.18 sec)
Records: 0  Duplicates: 0  Warnings: 0
```

Bien sûr, d'un point de vue fonctionnel, il n'y a aucune raison pour que **Client** possède une clé étrangère sur **Article**. Il ne s'agissait que d'un exemple et nous allons rectifier notre table. Par contre, comme il s'agit d'un champ particulier, vous ne pourrez pas effacer la colonne directement par **DROP COLUMN**. Il faudra exécuter :

```
root@localhost [siteMarchand]> ALTER TABLE Client DROP FOREIGN KEY
fk_idArticle;
Query OK, 0 rows affected (0.22 sec)
Records: 0  Duplicates: 0  Warnings: 0

root@localhost [siteMarchand]> ALTER TABLE Client DROP COLUMN fk_
idArticle;
Query OK, 0 rows affected (0.21 sec)
Records: 0  Duplicates: 0  Warnings: 0
```

Enfin, dans le cas d'une modification d'une base de données, la commande **ALTER DATABASE** permet de modifier les caractéristiques profondes de la base (encodage des caractères, etc.).

2 DML ou manipulation des données

Après avoir travaillé sur la structure, nous pouvons maintenant manipuler les données.

2.1 INSERT

La commande **INSERT** va permettre d'ajouter des enregistrements (des lignes). Ces ajouts vont pouvoir être réalisés sous différentes formes : ajout d'un enregistrement en affectant des valeurs à tous les champs, ajout de plusieurs enregistrements, ou ajout d'un enregistrement pour lequel seuls certains champs possèdent une valeur.

Ajoutons un enregistrement dans la table **Client** :

```
root@localhost [siteMarchand]> INSERT INTO Client VALUES (
    -> 1, 'Jim', 'Shannon', 'Hadrosaurus street', 2522, 'Terra Nova',
'js@tn.com');
Query OK, 1 row affected (0.21 sec)
```

Notez que comme nous ne spécifions pas les champs pour lesquels nous donnons des valeurs, il faut affecter une valeur à chaque colonne, y compris la première colonne qui est pourtant en **AUTO_INCREMENT**. Utilisons cette fois cette caractéristique en indiquant que nous ne donnons des valeurs que pour les 6 derniers champs :

```
root@localhost [siteMarchand]> INSERT INTO Client(prenom, nom,
adresse, codePostal, ville, mail) VALUES (
    -> 'Laura', 'Reily', 'Stegosaurus street', 2523, 'Terra Nova',
'lr@tn.com');
Query OK, 1 row affected (0.05 sec)
```

La clé primaire **idClient** étant en **AUTO_INCREMENT**, la valeur **2** lui a été attribuée automatiquement.

Enfin, pour des ajouts multiples, il suffit de lister les tuples à insérer :

```
root@localhost [siteMarchand]> INSERT INTO Client(prenom, nom,
adresse, codePostal, ville, mail) VALUES
    -> ('Alicia', 'Washington', 'Pterausaurus street', 3005, 'Terra
Nova', 'aw@tn.com'),
    -> ('Malcolm', 'Wallace', 'Brachiosaurus road', 2684, 'Terra
Nova', 'mw@tn.com');
Query OK, 2 rows affected (0.05 sec)
```

Si vous ne respectez pas les contraintes associées à la table, vous obtiendrez un message d'erreur :

```
root@localhost [siteMarchand]> INSERT INTO Client VALUES (
    -> 4, 'Nathaniel', 'Taylor', 'Acceraptor avenue', 3004, 'Terra
Nova', 'nt@tn.com');
ERROR 1062 (23000): Duplicate entry '4' for key 'PRIMARY'
```

Ici, comme la clé primaire 4 existe déjà (client Malcolm Wallace), il est impossible d'ajouter l'enregistrement.

2.2 DELETE

Pour supprimer un enregistrement, il va falloir être en mesure de l'identifier. Cela se fait à l'aide d'une clause **WHERE**, qui sera définie en détail dans la partie 4 relative à l'interrogation des données. Pour l'instant, nous allons l'utiliser en tant que simple condition identifiant un enregistrement. En utilisant la clé primaire, si **idClient = 4**, alors nous sommes sur la quatrième ligne de la table **Client**. Voici comment effacer cette ligne :

```
root@localhost [siteMarchand]> DELETE FROM Client WHERE idClient = 4;
Query OK, 1 row affected (0.05 sec)
```

Là encore, en fonction des contraintes, la suppression peut échouer et provoquer l'affichage d'un message d'erreur.

2.3 UPDATE

Pour modifier un enregistrement, on utilise la commande **UPDATE**. Comme avec **DELETE**, il faut cibler le ou les enregistrement(s) à modifier. En général, on utilise la clause **WHERE** :

```
root@localhost [siteMarchand]> UPDATE Client SET prenom = "Elisabeth" WHERE nom = "Shannon";
Query OK, 1 row affected (0.06 sec)
Rows matched: 1  Changed: 1  Warnings: 0
```

Ici, le premier enregistrement (relatif à Jim Shanon) est devenu :

```
root@localhost [siteMarchand]> SELECT * FROM Client;
...
|       1 | Elisabeth | Shannon    | Hadrosaurus street  |   2522 | Terra Nova | js@tn.com |
...
```

On peut aussi utiliser la clause **LIMIT** pour effectuer des modifications sur les **n** premiers enregistrements :

```
root@localhost [siteMarchand]> UPDATE Client SET CodePostal = 1111 LIMIT 3;
Query OK, 3 rows affected (0.05 sec)
Rows matched: 3  Changed: 3  Warnings: 0

root@localhost [siteMarchand]> SELECT * FROM Client;
+----------+----------+-----------+--------------------+-----------+-----------+----------+
| idClient | prenom   | nom       | adresse            | codePostal | ville     | mail     |
+----------+----------+-----------+--------------------+-----------+-----------+----------+
|       1 | Elisabeth | Shannon   | Hadrosaurus street |       1111 | Terra Nova | js@tn.com |
|       2 | Laura     | Reily     | Stegosaurus street |       1111 | Terra Nova | lr@tn.com |
|       3 | Alicia    | Washington | Pterausaurus street |       1111 | Terra Nova | aw@tn.com |
|       4 | Malcolm   | Wallace   | Brachiosaurus road |       2684 | Terra Nova | mw@tn.com |
+----------+----------+-----------+--------------------+-----------+-----------+----------+
4 rows in set (0.00 sec)
```

Bien sûr, il est également possible de modifier plusieurs colonnes d'un même enregistrement en une seule opération :

```
root@localhost [siteMarchand]> UPDATE Client SET prenom = "Jim", codePostal = 2522 WHERE nom
= "Shannon";
Query OK, 1 row affected (0.06 sec)
Rows matched: 1  Changed: 1  Warnings: 0
```

On peut enfin combiner les commandes **INSERT** et **UPDATE** pour modifier une colonne si une tentative d'ajout d'un enregistrement échoue à cause d'une clé primaire existante :

```
root@localhost [siteMarchand]> INSERT INTO Client VALUES (
    -> 4, 'Malcolm', 'Wallace', 'Brachiosaurus road', 2684, 'Terra Nova', 'mw@tn.com')
    -> ON DUPLICATE KEY UPDATE codePostal = codePostal + 1;
Query OK, 2 rows affected (0.05 sec)

root@localhost [siteMarchand]> SELECT * FROM Client;
...
|       4 | Malcolm | Wallace    | Brachiosaurus road  |      2685 | Terra Nova | mw@tn.com |
...
```

Ici, la nouvelle valeur du code postal est calculée à partir de l'ancienne valeur. Comme **codePostal** valait 2684, la nouvelle valeur **codePostal + 1** sera donc 2685.

3 DQL ou interrogation des données

La commande la plus couramment utilisée dans les requêtes SQL est la commande **SELECT**, qui permet d'extraire des informations d'une table (ou une vue). Nous avons déjà utilisé cette commande pour afficher tout le contenu d'une table grâce au caractère joker * qui indiquait que nous souhaitions récupérer toutes les colonnes. Si nous ne souhaitons que quelques informations particulières, il faudra le préciser. Voici un exemple où nous extrayons de la table **Client** les noms, prénoms et adresses mail :

```
root@localhost [siteMarchand]> SELECT
prenom, nom, mail FROM Client;
+---------+-----------+----------+
| prenom  | nom       | mail     |
+---------+-----------+----------+
| Jim     | Shannon   | js@tn.com |
| Laura   | Reily     | lr@tn.com |
| Alicia  | Washington | aw@tn.com |
| Malcolm | Wallace   | mw@tn.com |
...
+---------+-----------+----------+
8 rows in set (0.00 sec)
```

Si des noms de colonnes sont trop longs ou ne correspondent pas à ce qui est attendu, nous pouvons les modifier grâce à l'instruction **AS** :

```
root@localhost [siteMarchand]> SELECT
prenom AS firstname, nom AS name, mail FROM
Client;
+----------+-----------+----------+
| firstname | name      | mail     |
+----------+-----------+----------+
| Jim      | Shannon   | js@tn.com |
...
+----------+-----------+----------+
8 rows in set (0.00 sec)
```

Nous avons vu la base de la commande **SELECT**. Associée à d'autres instructions, nous pouvons modifier les résultats transmis. Ce sont ces instructions que nous allons voir dans la suite.

3.1 Informations distinctes

Imaginons maintenant que nous ajoutions un client portant le nom d'un client déjà référencé (table **Client**) : voir tableau ci-contre.

Si nous souhaitons afficher la liste des noms des clients, nous obtiendrons un doublon :

idClient	prenom	nom	adresse	codePostal	ville	mail
9	Elisabeth	Shannon	Hadrosaurus street	2522	TerraNova	js@tn.com


```
root@localhost [siteMarchand]> SELECT nom
FROM Client;
+-----------+
| nom       |
+-----------+
...
| Shannon   |
| Shannon   |
...
+-----------+
9 rows in set (0.00 sec)
```

Les instructions **DISTINCT** et **DISTINCTROW** permettent l'élimination des colonnes (ou lignes pour la seconde instruction) dupliquées. Donc, pour n'obtenir aucune répétition dans la liste des noms des clients, il nous faut modifier légèrement notre requête :

```
root@localhost [siteMarchand]> SELECT
DISTINCT(nom) FROM Client;
+-----------+
| nom       |
+-----------+
...
| Shannon   |
...
+-----------+
9 rows in set (0.00 sec)
```

3.2 Concaténation

L'opérateur de concaténation va nous permettre d'associer des informations de n'importe quel type pour produire une chaîne de caractères. Nous pouvons par exemple obtenir une phrase indiquant où habitent les clients :

```
root@localhost [siteMarchand]> SELECT
CONCAT(prenom, ' ', nom, ' habite a ', adresse) AS
phrase FROM Client;
+-----------------------------------------+
| phrase                                  |
+-----------------------------------------+
| Jim Shannon habite a Hadrosaurus street |
| Laura Reily habite a Stegosaurus street |
...
| Elisabeth Shannon habite a Hadrosaurus street |
+-----------------------------------------+
9 rows in set (0.00 sec)
```

3.3 Limiter le nombre de lignes du résultat

Souvent, lorsqu'une table contient beaucoup de données, nous n'avons pas besoin d'extraire l'ensemble des enregistrements sur une requête. La clause **LIMIT** va permettre de n'extraire que **n** lignes à partir d'une ligne de départ donnée (la première ligne d'une table a pour indice 0). On doit donc donner **LIMIT depart, nb_ligne** (par défaut, si **depart** est omis, on considère qu'il s'agit de la ligne 0). Voici un exemple sur la table **Client** où l'on ne récupère que les deuxième et troisième enregistrements :

```
root@localhost [siteMarchand]> SELECT * FROM Client LIMIT 1, 2;
+----------+--------+-----------+--------------------+-----------+-----------+----------+
| idClient | prenom | nom       | adresse            | codePostal | ville     | mail     |
+----------+--------+-----------+--------------------+-----------+-----------+----------+
|        2 | Laura  | Reily     | Stegosaurus street |       1111 | Terra Nova | lr@tn.com |
|        3 | Alicia | Washington | Pterausaurus street |       1111 | Terra Nova | aw@tn.com |
+----------+--------+-----------+--------------------+-----------+-----------+----------+
2 rows in set (0.00 sec)
```

3.4 Ordonner les résultats

Pour trier les résultats suivant une colonne en ordre descendant ou ascendant, on utilise l'instruction **ORDER BY** suivie du nom de la colonne et éventuellement de l'ordre du tri : **ASC** pour ascendant (choix par défaut) ou **DESC** pour descendant. Affichons par exemple les prénoms et noms des clients triés par ordre alphabétique inverse sur les prénoms :

```
root@localhost [siteMarchand]> SELECT prenom, nom FROM Client ORDER BY prenom DESC;
+----------+-----------+
| prenom   | nom       |
+----------+-----------+
| Skye     | Tate      |
...
| Alicia   | Washington |
+----------+-----------+
9 rows in set (0.00 sec)
```

3.5 Les calculs

Il existe de nombreuses fonctions de calculs mathématiques, de modifications de chaînes, de manipulations de dates, etc. On peut par exemple additionner des valeurs numériques de colonnes avec **SUM**, calculer une moyenne avec **AVG**, compter le nombre d'enregistrements avec **COUNT**, obtenir la longueur d'une chaîne avec **LENGTH**, passer tous les caractères d'une chaîne en minuscules avec **LOWER**, ajouter un certain nombre de jours à une date avec **ADDDATE**, etc. Il y a beaucoup trop de fonctions pour toutes les énumérer ici. La page de documentation du SGBD que vous utilisez vous apportera toutes les informations voulues (pour MySQL voir **[4]**).

Voici par exemple comment calculer le nombre de clients :

```
root@localhost [siteMarchand]> SELECT COUNT(*) AS Nb_Clients FROM Client;
+------------+
| Nb_Clients |
+------------+
|          9 |
+------------+
1 row in set (0.00 sec)
```

Maintenant, si nous souhaitons obtenir un décompte détaillé avec le nombre de clients portant le même nom, il va falloir grouper les résultats.

3.6 Grouper les résultats

La clause **GROUP BY** va permettre de grouper les résultats d'un calcul suivant les valeurs d'une colonne. En suivant notre exemple précédent, si nous groupons les résultats par nom, nous allons obtenir une liste de noms et le nombre de clients qui les portent :

```
root@localhost [siteMarchand]> SELECT nom,
COUNT(*) AS Nb_Clients FROM Client GROUP
BY(nom);
+-----------+-----------+
| nom       | Nb_Clients |
+-----------+-----------+
| Reily     |          1 |
| Shannon   |          2 |
...
| Washington |          1 |
+-----------+-----------+
9 rows in set (0.00 sec)
```

Si nous souhaitons conditionner les résultats, nous pourrons utiliser la clause **HAVING** qui vérifie une condition sur les résultats obtenus par le **GROUP BY** :

```
root@localhost [siteMarchand]> SELECT nom,
COUNT(*) AS Nb_Clients FROM Client GROUP
BY(nom) HAVING(Nb_Clients = 1);
+-----------+-----------+
| nom       | Nb_Clients |
+-----------+-----------+
| Reily     |          1 |
...
| Washington |          1 |
+-----------+-----------+
8 rows in set (0.00 sec)
```

Ici, nous n'avons extrait que les noms des clients qui n'apparaissent qu'une fois dans la base.

3.7 La clause WHERE

Grâce à la clause **WHERE**, nous pouvons restreindre les informations recherchées par une requête grâce à des conditions composées des opérateurs suivants :

- Opérateurs de comparaison : **=**, **<>** (différent), **<**, **>**, **<=** (inférieur ou égal), **>=** (supérieur ou égal).
- Opérateurs logiques permettant de créer des conditions complexes : **OR**, **AND**, **NOT**.
- Opérateurs « SQL » :
  - **LIKE** : teste la concordance de chaînes de caractères. Cet opérateur autorise l'utilisation de caractères joker : **_** remplace un caractère et **%** remplace au moins un caractère ;
  - **IS NULL** : teste si un champ n'a pas été affecté (et qu'il est donc égal à **NULL**) ;
  - **BETWEEN** : teste l'appartenance à un intervalle. **BETWEEN 1 AND 12** teste si une valeur est comprise entre 1 et 12 ;
  - **IN** : teste l'appartenance à un groupe de valeurs données sous forme de liste. **IN (2, 4, 6)** teste si une valeur est égale à 2, 4, ou 6.

Nous pouvons ainsi rechercher les clients dont le code postal est entre 2000 et 3000, qui habitent une rue (street) et dont le nom se trouve dans la liste (Shannon, Reily, Wallace, Boylan) :

```
root@localhost [siteMarchand]> SELECT * FROM Client WHERE
    -> codePostal BETWEEN 2000 AND 3000 AND
    -> adresse LIKE "% street" AND
    -> nom IN ("Shannon", "Reily", "Wallace", "Boylan");
+----------+----------+---------+------------------+------------+-----------+----------+
| idClient | prenom   | nom     | adresse          | codePostal | ville     | mail     |
+----------+----------+---------+------------------+------------+-----------+----------+
|        1 | Jim      | Shannon | Hadrosaurus street |       2522 | Terra Nova | js@tn.com |
|        2 | Laura    | Reily   | Stegosaurus street |       2523 | Terra Nova | lr@tn.com |
|        9 | Elisabeth | Shannon | Hadrosaurus street |       2522 | Terra Nova | es@tn.com |
+----------+----------+---------+------------------+------------+-----------+----------+
3 rows in set (0.00 sec)
```

3.8 La combinaison de résultats

La clause **UNION** permet de combiner les résultats de plusieurs appels à **SELECT** (dans le cas où une condition serait plus longue à écrire par exemple). Attention toutefois à l'utilisation de cette commande : les noms de colonnes des résultats ne sont extraits que du premier **SELECT**... Donc si vos différentes requêtes n'extraient pas les mêmes colonnes, vous risquez d'avoir des résultats assez difficilement utilisables :

```
root@localhost [siteMarchand]> SELECT nom, prenom FROM Client WHERE codePostal < 2000 UNION
    -> SELECT nom, adresse FROM Client WHERE nom LIKE "S%";
+-----------+------------------+
| nom       | prenom           |
+-----------+------------------+
| Washington | Alicia           |
| Shannon   | Hadrosaurus street |
+-----------+------------------+
2 rows in set (0.00 sec)
```

Dans le résultat de la requête précédente, la deuxième ligne pose problème : « Hadrosaurus street » est visiblement une adresse, or elle apparaît en tant que « prenom »... Cela est dû au fait que le premier **SELECT** extrait les colonnes **nom** et **prenom**, alors que le second extrait **nom** et **adresse**.

3.9 Les jointures

Les jointures permettent d'extraire des données issues de plusieurs tables. Depuis le début de cet article, nous ne travaillons qu'avec la table **Client**, nous aurons ici besoin de peupler les tables **Article** et **Commande**. **Article** contiendra les données suivantes :

IdArticle	designation	puHT
1	Plex	1024.99
2	Batterie	5557.86
3	Mine sonique	12000.00

Deux commandes ont été référencées dans la table **Commande** :

idCommande	fk_idArticle	fk_idClient
1	2	9
2	3	5

Ce tableau indique qu'Elisabeth Taylor (clé primaire 9 de la table **Client**) a commandé une batterie (clé primaire 2 de la table **Article**) et que Nathaniel Taylor (clé primaire 5 de la table **Client**) a commandé une mine sonique (clé primaire 3 de la table **Article**).

Il existe plusieurs types de jointures que nous allons détailler.

3.9.1 La jointure interne

Il s'agit d'une jointure qui s'opère sur des conditions de la clause **WHERE** et on peut donc l'écrire sans nouvelle commande. Pour connaître le prénom et le nom des clients ayant passé une commande, il faudra exécuter :

```
root@localhost [siteMarchand]> SELECT prenom, nom FROM
    -> Client, Commande WHERE
    -> idClient = fk_idClient;
+----------+--------+
| prenom   | nom    |
+----------+--------+
| Jim      | Shannon |
| Nathaniel | Taylor |
+----------+--------+
2 rows in set (0.00 sec)
```

Lorsque des tables interviennent dans une même requête et possèdent des noms de colonnes identiques, il faut leur donner un nom à l'intérieur de la requête :

```
root@localhost [siteMarchand]> SELECT prenom, nom
    -> FROM Client c, Commande co
    -> WHERE c.idClient = co.fk_idClient;
...
```

Cette jointure peut également s'écrire à l'aide de la clause **JOIN** ou **INNER JOIN**. En effet, par défaut, une jointure est interne (**INNER**) et vous pourrez utiliser indifféremment l'une ou l'autre. Notre requête précédente peut donc s'écrire :

```
root@localhost [siteMarchand]> SELECT prenom, nom FROM Client
    -> JOIN Commande
    -> ON idClient = fk_idClient;
...
```

Bien sûr notre requête était ici très simple, mais on peut réaliser des requêtes complexes qui sont très esthétiques et qui se lisent bien (à condition de rester très rigoureux !). Par exemple, voici une requête affichant sous forme de phrase la liste des personnes ayant commandé un article et l'article commandé :

```
root@localhost [siteMarchand]> SELECT CONCAT(c.prenom, ' ', c.nom,
' a commande ', a.designation) AS 'Liste des commandes'
    -> FROM Client c
    -> JOIN Commande co ON co.fk_idClient = c.idClient
    -> JOIN Article a ON co.fk_idArticle = a.idArticle;
+-----------------------------------+
| Liste des commandes               |
+-----------------------------------+
| Elisabeth Shannon a commande Batterie |
| Nathaniel Taylor a commande Mine sonique  |
+-----------------------------------+
2 rows in set (0.00 sec)
```

3.9.2 La jointure externe

La jointure externe est une jointure qui respecte les mêmes conditions que la jointure interne, mais qui ajoute des lignes ne respectant pas la condition de la table se situant :

- à gauche (avant le **JOIN**) : **LEFT OUTER JOIN** (ou simplement **LEFT JOIN** puisqu'à partir du moment où il y a une précision sur la jointure, celle-ci est externe (**OUTER**)) ;
- à droite (après le **JOIN**) : **RIGHT OUTER JOIN** ou **RIGHT JOIN** ;
- à gauche et droite : **FULL OUTER JOIN** ou **FULL JOIN**.

Si nous reprenons la première jointure que nous avons effectuée **(SELECT prenom, nom FROM Client JOIN Commande ON idClient = fk_idClient;)** et que nous la transformons en jointure externe gauche, nous allons ajouter les lignes de la table **Client** qui ne respectent pas la condition de jointure... Nous aurons donc la liste de tous les clients :

```
root@localhost [siteMarchand]> SELECT prenom, nom FROM Client LEFT
OUTER JOIN Commande ON idClient = fk_idClient;
+----------+-----------+
| prenom   | nom       |
+----------+-----------+
| Jim      | Shannon   |
...
+----------+-----------+
9 rows in set (0.00 sec)
```

4 DCL ou contrôle des données

Tous les exemples que nous avons exécutés jusqu'à présent ont été lancés par l'utilisateur **root**. Cet utilisateur disposant de tous les droits, il a accès à toutes les commandes... Mais il est possible de définir des droits pour les utilisateurs.

4.1 GRANT et REVOKE

La commande **GRANT** permet justement d'attribuer des privilèges aux utilisateurs. Pour visualiser les privilèges d'un utilisateur donné, vous pourrez utiliser la commande **SHOW** :

```
root@localhost [siteMarchand]> SHOW GRANTS FOR 'root'@'localhost';
+-----------------------------------------------------------+
| Grants for root@localhost                                 |
+-----------------------------------------------------------+
| GRANT ALL PRIVILEGES ON *.* TO 'root'@'localhost' IDENTIFIED BY
PASSWORD '*D8562F27...671' WITH GRANT OPTION |
| GRANT PROXY ON ''@'' TO 'root'@'localhost' WITH GRANT OPTION    |
+-----------------------------------------------------------+
2 rows in set (0.00 sec)
```

Lors de l'ajout d'un utilisateur, on spécifie ses privilèges à l'aide de cette commande. Dans la partie traitant de la commande **CREATE** nous n'avons pas abordé la création d'un utilisateur, car il nous manquait la partie d'attribution des privilèges. Nous pouvons donc maintenant créer un nouvel utilisateur :

```
root@localhost [siteMarchand]> CREATE USER 'tristan'@'localhost'
IDENTIFIED BY 'monMotDePasse';
Query OK, 0 rows affected (0.00 sec)
```

Nous avons spécifié ici le nom de l'utilisateur, l'adresse du serveur de base de données (**localhost**) et le mot de passe à employer pour se connecter. Nous pouvons maintenant lui donner des privilèges sur certaines bases, ou seulement certaines tables. Les privilèges sont spécifiés par le nom des commandes qui seront autorisées : **CREATE**, **CREATE USER**, **SELECT**, **DROP**, etc. Il y a 4 privilèges un peu particuliers :

- **ALL** : donne tous les privilèges ;
- **ALTER** : donne le droit de modifier la structure de la base ou de la table ;
- **SUPER** : donne le droit de gérer les déclencheurs (que nous verrons plus loin) ;
- **USAGE** : pas de privilèges.

Nous allons par exemple donner à notre nouvel utilisateur le droit d'extraire des données de la base **siteMarchand**, mais de n'insérer des données que dans la table **Client** :

```
root@localhost [siteMarchand]> GRANT SELECT
    -> ON siteMarchand .*
    -> TO 'tristan'@'localhost';
Query OK, 0 rows affected (0.00 sec)

root@localhost [siteMarchand]> GRANT INSERT
    -> ON siteMarchand.Client
    -> TO 'tristan'@'localhost';
Query OK, 0 rows affected (0.00 sec)
```

En se déconnectant et en se reconnectant en tant qu'utilisateur **tristan**, nous pouvons bien extraire des données des tables, ajouter des clients, mais lorsque l'on tente d'ajouter une commande, un message d'erreur nous avertit que nous ne disposons pas des droits suffisants :

```
tristan@localhost [siteMarchand]> INSERT INTO Commande VALUES (1, 3,
1);
ERROR 1142 (42000): INSERT command denied to user
'tristan'@'localhost' for table 'Commande'
```

La commande **REVOKE** est l'inverse de la commande **GRANT** : elle permet de révoquer, supprimer un privilège à un utilisateur. Elle s'utilise exactement de la même manière que **GRANT**.

4.2 COMMIT et ROLLBACK

Une transaction est un ensemble d'instructions de manipulation des données (DML), qui fait passer la base de données d'un état à un autre tout en maintenant la stabilité du système. Lorsqu'une transaction s'exécute sans erreur elle peut être validée ou invalidée. Dans le premier cas, les modifications sont alors inscrites dans la base, alors que dans le second elles sont simplement « oubliées ». La commande permettant de valider une transaction est **COMMIT** et celle permettant d'annuler une transaction est **ROLLBACK**.

Le moteur InnoDB de MySQL fonctionne par défaut en mode de validation automatique : impossible d'annuler une transaction. Pour pouvoir comprendre comment fonctionne le système des transactions, il faut annuler la validation automatique :

```
root@localhost [siteMarchand]> SET AUTOCOMMIT = 0;
Query OK, 0 rows affected (0.00 sec)
```

Bien sûr, si vous souhaitez réactiver la validation automatique, il vous suffit d'affecter la valeur **1** à **AUTOCOMMIT**.

Essayons maintenant de modifier une valeur dans la table **Article** :

```
root@localhost [siteMarchand]> UPDATE Article SET designation =
'Batterie pour Plex' WHERE idArticle = 2;
Query OK, 1 row affected (0.02 sec)
Rows matched: 1  Changed: 1  Warnings: 0

root@localhost [siteMarchand]> SELECT * FROM Article;
+----------+------------------+--------+
| idArticle | designation      | puHT   |
+----------+------------------+--------+
|         1 | Plex             | 1024.99 |
|         2 | Batterie pour Plex | 5557.86 |
|         3 | Mine sonique     |  12000 |
+----------+------------------+--------+
3 rows in set (0.00 sec)
```

Comme vous pouvez le voir sur le résultat du **SELECT**, la modification a bien été effectuée... Mais alors quelle est la différence avec la validation automatique ? Quittez MySQL et relancez le **SELECT**... La table est revenue à son état antérieur : la transaction n'avait pas été validée !

Recommençons notre manipulation et essayons à nouveau de modifier un article après avoir désactivé la validation automatique. En effet, la validation automatique est réactivée à chaque démarrage. Pour le vérifier, vous pouvez taper la commande suivante :

```
root@localhost [siteMarchand]> SHOW GLOBAL VARIABLES LIKE "autocommit";
+--------------+-------+
| Variable_name | Value |
+--------------+-------+
| autocommit    | ON    |
+--------------+-------+
1 row in set (0.00 sec)
```

Donc, une fois la validation automatique désactivée et la modification effectuée, vous pouvez soit la valider, soit l'annuler. Ici, nous allons annuler la modification avec **ROLLBACK** :

```
root@localhost [siteMarchand]> ROLLBACK;
Query OK, 0 rows affected (0.04 sec)

root@localhost [siteMarchand]> SELECT * FROM Article;
+-----------+-------------+---------+
| idArticle | designation  | puHT    |
+-----------+-------------+---------+
|         1 | Plex         | 1024.99 |
|         2 | Batterie     | 5557.86 |
|         3 | Mine sonique |   12000 |
+-----------+-------------+---------+
3 rows in set (0.00 sec)
```

Comme prévu, nous revenons à la valeur antérieure de la table **Article**. Si nous avions voulu valider la modification, il aurait fallu exécuter **COMMIT;**.

Nous pouvons enchaîner des transactions et créer des points de validation à l'aide de l'instruction **SAVEPOINT**. Ces points de validation portent une étiquette et la validation ou l'annulation pourront être exécutées jusqu'à une étiquette donnée. Voici un exemple où nous modifions 3 fois la même valeur en insérant des points de validation. Nous pouvons ainsi annuler progressivement des transactions :

```
root@localhost [siteMarchand]> UPDATE Article SET puHT = 35 WHERE idArticle = 2;
Query OK, 1 row affected (0.01 sec)
Rows matched: 1  Changed: 1  Warnings: 0

root@localhost [siteMarchand]> SAVEPOINT P1;
Query OK, 0 rows affected (0.00 sec)

root@localhost [siteMarchand]> UPDATE Article SET puHT = 43 WHERE idArticle = 2;
Query OK, 1 row affected (0.00 sec)
Rows matched: 1  Changed: 1  Warnings: 0

root@localhost [siteMarchand]> SAVEPOINT P2;
Query OK, 0 rows affected (0.00 sec)

root@localhost [siteMarchand]> UPDATE Article SET puHT = 55 WHERE idArticle = 2;
Query OK, 1 row affected (0.00 sec)
Rows matched: 1  Changed: 1  Warnings: 0
```

Si nous affichons la table **Article**, la valeur du prix pour le second article est bien sûr 55 :

```
root@localhost [siteMarchand]> SELECT
* FROM Article;
+-----------+-------------+---------+
| idArticle | designation  | puHT    |
+-----------+-------------+---------+
|         1 | Plex         | 1024.99 |
|         2 | Batterie     |      55 |
|         3 | Mine sonique |   12000 |
+-----------+-------------+---------+
3 rows in set (0.00 sec)
```

Grâce aux points de validation, nous pouvons remonter dans l'historique pour ne valider que les modifications qui nous satisfont. Nous allons remonter jusqu'au point **P1** et valider les modifications :

```
root@localhost [siteMarchand]> ROLLBACK TO
SAVEPOINT P1;
Query OK, 0 rows affected (0.00 sec)

root@localhost [siteMarchand]> SELECT *
FROM Article;
+-----------+-------------+---------+
| idArticle | designation  | puHT    |
+-----------+-------------+---------+
|         1 | Plex         | 1024.99 |
|         2 | Batterie     |      35 |
|         3 | Mine sonique |   12000 |
+-----------+-------------+---------+
3 rows in set (0.00 sec)

root@localhost [siteMarchand]> COMMIT;
Query OK, 0 rows affected (0.05 sec)
```

Notez qu'avec le moteur InnoDB, si vous souhaitez ne pas modifier la valeur **AUTOCOMMIT** mais bénéficier tout de même du système de validation / annulation des transactions, vous pouvez débuter vos transactions par : **START TRANSACTION;**. À ce moment-là, le fonctionnement se fera de la même manière qu'avec un **AUTOCOMMIT** à **0**.

5 Les fonctions et les procédures

Il est possible de déclarer des blocs de commandes SQL sous la forme de fonctions ou de procédures (fonctions ne renvoyant aucune valeur directement). Les blocs sont délimités par les instructions **BEGIN** et **END**.

Pour déclarer une fonction ou une procédure dans la console MySQL,

nous allons rencontrer un petit problème : le délimiteur **;** étant la fin de la requête, nous aurons très rapidement un message d'erreur. Il va donc falloir commencer par modifier le délimiteur grâce à la commande **DELIMITER**. Voici une déclaration de fonction qui compte la somme du prix des articles dont le prix est inférieur ou égal à n dans la table **Article** :

```
root@localhost [siteMarchand]> DELIMITER //
root@localhost [siteMarchand]> CREATE FUNCTION totalPrix(n INTEGER)
RETURNS DOUBLE
    -> BEGIN
    ->     DECLARE resultat DOUBLE;
    ->     SELECT SUM(puHT) INTO resultat FROM Article WHERE puHT <= n;
    ->     RETURN resultat;
    -> END //
Query OK, 0 rows affected (0.02 sec)

root@localhost [siteMarchand]> DELIMITER ;
```

La première ligne permet de désactiver le caractère **;** en tant que fin d'instruction et de le remplacer par **//**. La ligne suivante déclare une fonction **totalPrix** qui accepte un paramètre de type entier et qui renverra un résultat sous la forme d'un réel de type **DOUBLE**.

Le corps de la fonction est compris entre **BEGIN** et **END** (notez la fin d'instruction sous la forme **//**). Dans cette fonction, nous utilisons une variable locale **resultat** qui est déclarée avec l'instruction **DECLARE**. Cette variable va contenir le résultat de notre requête grâce à un usage un peu particulier de **SELECT** : **SELECT champs INTO variable FROM table**. L'instruction **RETURN** permet de renvoyer le résultat de notre requête contenu dans la variable **resultat**. Une fois la fonction déclarée, nous réactivons le délimiteur par défaut.

Maintenant que la fonction est déclarée, il faut l'appeler. Cela se fait en utilisant un **SELECT** :

```
root@localhost [siteMarchand]> SELECT totalPrix(10000);
+-----------------+
| totalPrix(10000) |
+-----------------+
|         1059.99 |
+-----------------+
1 row in set (0.00 sec)
```

Ici, nous n'avons fait qu'afficher le résultat de la fonction, mais elle peut tout à fait être utilisée à l'intérieur d'une clause **WHERE** pour déterminer une condition.

Vous avez pu voir que l'écriture de fonctions n'était pas très évidente dans la console. Si vous le souhaitez, vous pouvez placer le code de vos fonctions et procédures dans un fichier externe que vous pourrez charger grâce à la commande **SOURCE** :

```
root@localhost [siteMarchand]> SOURCE monFichier.sql;
Query OK, 0 rows affected (0.00 sec)
```

L'écriture des procédures est très proche de celle des fonctions. Au niveau des différences, il faut indiquer si les paramètres sont utilisés en entrée (**IN**), en sortie (modification de la valeur pour usage à l'extérieur de la fonction avec **OUT**), ou en entrée/sortie (**INOUT**). De plus, l'appel à une procédure se fait par la commande **CALL**.

Voici l'écriture sous forme de procédure de la fonction précédente. Ce code sera placé dans un fichier **fct.sql** :

```
01: DELIMITER //
02:
03: CREATE PROCEDURE totalPrix_Proc(IN n INTEGER, OUT resultat DOUBLE)
04:     BEGIN
05:         SELECT SUM(puHT) INTO resultat FROM Article WHERE puHT <= n;
06:     END //
07:
08: DELIMITER ;
```

Pour utiliser ce code, nous aurons besoin d'une variable pouvant contenir le résultat. La déclaration d'une variable « générique » se fait grâce à la commande **SET** et le nom de la variable est préfixé par **@**. Voici comment charger notre code et l'exécuter :

```
root@localhost [siteMarchand]> source fct.sql;
Query OK, 0 rows affected (0.00 sec)

root@localhost [siteMarchand]> SET @resultat = '';
Query OK, 0 rows affected (0.00 sec)

root@localhost [siteMarchand]> CALL totalPrix_Proc(10000, @resultat);
Query OK, 1 row affected (0.00 sec)

root@localhost [siteMarchand]> SELECT @resultat;
+----------+
| @resultat |
+----------+
|   1059.99 |
+----------+
1 row in set (0.00 sec)
```

6 Les curseurs

Les curseurs sont énormément utilisés par les API des différents langages interagissant avec un SGBD. Un curseur est le résultat d'un **SELECT** mis en cache côté serveur et permettant de parcourir les lignes une à une. Dans MySQL, les curseurs ne sont supportés qu'à l'intérieur de procédures ou fonctions. Au niveau du processus d'utilisation des curseurs, il faut suivre 4 étapes :

- Déclaration du curseur contenant la requête ;
- Ouverture du curseur : exécution de la requête et chargement des lignes ;

- Parcours des lignes ;

- Fermeture du curseur.

Voici un exemple d'utilisation où nous allons suivre ces étapes pour afficher les lignes de la table **Article** :

```
01: DELIMITER //
02:
03: CREATE PROCEDURE teste_Curseur()
04:     BEGIN
05:         -- Déclaration de la variable qui permettra de lire les données
06:         -- du curseur
07:         DECLARE nom_article VARCHAR(100);
08:
09:         -- Déclaration du curseur
10:         DECLARE curs_1 CURSOR FOR
11:             SELECT designation FROM Article;
12:
13:         -- Ouverture du curseur
14:         OPEN curs_1;
15:
16:         -- Extraction des données du curseur
17:         FETCH curs_1 INTO nom_article;
18:         SELECT nom_article;
19:
20:         -- Fermeture du curseur
21:         CLOSE curs_1;
22:     END //
23:
24: DELIMITER ;
```

J'ai introduit dans ce code des commentaires : tous les caractères qui suivent **--** ne sont pas considérés comme du code. Lorsque nous utilisons ce code, nous nous apercevons que nous n'avons extrait que la première ligne des articles :

```
root@localhost [siteMarchand]> CALL teste_Curseur();
+-------------+
| nom_article |
+-------------+
| Plex        |
+-------------+
1 row in set (0.01 sec)
```

En fait c'est faux : nous avons bien extrait toutes les lignes, mais nous ne les avons pas toutes parcourues. Pour pouvoir créer une boucle, nous allons ajouter une variable booléenne et lui associer un changement de valeur dans le cas où le message **NOT FOUND** est envoyé.

Dans le code ci-dessous, les modifications permettant d'afficher toutes les lignes extraites sont présentées en rouge :

```
01: DELIMITER //
02:
03: CREATE PROCEDURE teste_Curseur()
04:     BEGIN
05:         -- Déclaration de la variable qui permettra de lire les
données
06:         -- du curseur
07:         DECLARE nom_article VARCHAR(100);
08:         DECLARE finBoucle BOOLEAN DEFAULT 0;
09:
10:         -- Déclaration du curseur
11:         DECLARE curs_1 CURSOR FOR
12:             SELECT designation FROM Article;
13:         DECLARE CONTINUE HANDLER FOR NOT FOUND
14:             SET finBoucle = 1;
15:
16:         -- Ouverture du curseur
17:         OPEN curs_1;
18:
19:         -- Extraction des données du curseur
20:         WHILE (NOT finBoucle) DO
21:             FETCH curs_1 INTO nom_article;
22:             IF NOT finBoucle THEN
23:                 SELECT nom_article;
24:             END IF;
25:         END WHILE;
26:
27:         -- Fermeture du curseur
28:         CLOSE curs_1;
29:     END //
30:
31: DELIMITER ;
```

Dans ce code, nous avons utilisé plusieurs nouvelles instructions. En ligne 8, nous avons ajouté une variable booléenne **finBoucle** initialisée à 0. Dans les lignes 13 et 14, nous déclarons un gestionnaire de traitement des erreurs : si l'erreur **NOT FOUND** est détectée (tous les codes d'erreur commençant par 02), alors la variable **finBoucle** passe à 1. En ligne 20, nous débutons notre boucle qui va lire chaque ligne du curseur jusqu'à ce que **finBoucle** soit égale à 1 et en ligne 22, nous testons la valeur de **finBoucle** pour savoir si nous affichons ou pas le résultat (sans cette boucle la dernière ligne est affichée 2 fois). Les lignes 24 et 25 referment les blocs de test et de boucle.

Cette fois, en exécutant le code, toutes les lignes apparaissent correctement :

```
root@localhost [siteMarchand]> CALL teste_Curseur();
+-------------+
| nom_article |
+-------------+
| Plex        |
+-------------+
1 row in set (0.00 sec)
...
+--------------+
| nom_article  |
+--------------+
| Mine sonique |
+--------------+
1 row in set (0.00 sec)

Query OK, 0 rows affected, 1 warning (0.00 sec)
```

7 Les déclencheurs

Les déclencheurs, aussi appelés *triggers*, sont des bouts de code SQL programmés pour se déclencher automatiquement lors d'une action particulière. Pour être plus

précis, ils se déclenchent avant (**BEFORE**) ou après (**AFTER**) une action. Nous pouvons ainsi créer un trigger qui se déclenche après l'ajout d'une ligne dans la table **Client** et qui augmente le prix des articles de la table **Article** :

```
root@localhost [siteMarchand]> DELIMITER //
root@localhost [siteMarchand]> CREATE TRIGGER augmenteAutoArticle
    -> AFTER INSERT ON Client
    -> FOR EACH ROW
    -> BEGIN
    ->     UPDATE Article SET puHT = puHT * 1.1;
    -> END //
Query OK, 0 rows affected (0.08 sec)

root@localhost [siteMarchand]> DELIMITER ;
```

Désormais, à chaque fois que nous allons ajouter un nouveau client, tous les articles qui étaient déjà présents dans la table subiront une augmentation de 10% :

```
root@localhost [siteMarchand]> INSERT INTO Client VALUES (10,
'Maddy', 'Shannon', 'Hadrosaurus street', 2522, 'Terra Nova',
'ms@tn.com');
Query OK, 1 row affected (0.05 sec)

root@localhost [siteMarchand]> SELECT * FROM Article;
+-----------+-------------+-------------------+
| idArticle | designation  | puHT              |
+-----------+-------------+-------------------+
|         1 | Plex        |           1127.489 |
|         2 | Batterie    |              38.5 |
|         3 | Mine sonique | 13200.000000000002 |
+-----------+-------------+-------------------+
3 rows in set (0.00 sec)
```

Notez que MySQL ne supporte pas les triggers déclenchés et agissant sur la même table (nous n'aurions pas pu modifier la table **Client**, puisque c'est sur une action sur celle-ci que le trigger est déclenché).

8 Les injections SQL

Je ne pouvais pas finir cet article sans un petit mot sur les injections SQL. Lors de ces intrusions, l'attaquant manipule des requêtes SQL et pourra corrompre la base de données. Les dégâts pourront aller de la falsification d'identité à la prise de contrôle de la base de données, en passant par la destruction ou la modification des données et de la structure de la base. Le principe est très simple : il faut, depuis un programme, transmettre du code SQL malveillant dans une zone de saisie censée recueillir une entrée utilisateur, cette zone étant non protégée. Le code SQL de l'attaquant est alors exécuté et lui permet de récupérer des informations ou de détériorer la base.

Imaginons que notre table **Client** possède un champ supplémentaire **password** contenant un mot de passe. Pour se connecter à un service, le client devra indiquer son adresse mail et son mot de passe. La requête permettant de vérifier si le client est présent dans la base et si le mot de passe est correct sera la suivante :

```
SELECT idClient FROM Client WHERE mail='monMail' AND
password=SHA1('monMotDePasse');
```

Si l'attaquant, lors de la saisie de l'adresse mail, inscrit **' OR 1 = 1; --**, la requête SQL va être modifiée pour aboutir à :

```
SELECT idClient FROM Client WHERE mail='' OR 1 = 1; --' AND
password=SHA1('monMotDePasse');
```

Tout le code indiqué en rouge ne sera pas exécuté, puisque se trouvant en commentaire, et en logique booléenne, x OR True = True (en français, « n'importe quoi » ou VRAI vaut VRAI). La valeur **1=1** valant toujours VRAI, dans notre requête, l'assertion de la clause **WHERE** est toujours VRAI (x OR 1=1 (VRAI)), le résultat sera l'identifiant du premier utilisateur rencontré et l'authentification sera validée !

Dans vos programmes, lorsque vous utilisez des saisies utilisateur pour interagir avec une base de données, considérez-les toujours comme potentiellement malveillantes ! Pour contrer cette attaque, il existe des précautions très simples à prendre : échappez les caractères spéciaux apostrophe, guillemet, etc. Reconnaissez qu'il serait dommage de perdre des données pour des techniques aussi simples à mettre en œuvre...

Conclusion

Le langage SQL est le cœur des SGBD, puisque c'est lui qui nous permet de commander le système. Il est capable de réaliser de nombreuses opérations et, même s'il est possible d'extraire des données de manière grossière pour les retraiter par la suite dans un autre langage, il est bien souvent plus rapide de traiter directement les données lors de la requête (au moins pour les dégrossir). Ainsi, que ce soit en C, Python, Java ou autre, à moins d'utiliser un ORM (*Object-Relational Mapping* définissant des liaisons entre objets et tables), lorsque vous communiquerez avec un SGBD, vous devrez écrire du SQL. ■

Références

[1] Respect des normes SQL dans MySQL : http://dev.mysql.com/doc/refman/5.7/en/compatibility.html

[2] Respect des normes SQL dans PostgreSQL : http://www.postgresql.org/docs/9.2/static/features.html

[3] Manuel de référence MySQL : http://dev.mysql.com/doc/refman/5.7/en/index.html

[4] Liste des fonctions MySQL classées par type : http://dev.mysql.com/doc/refman/5.7/en/functions.html

GNU LINUX MAGAZINE / FRANCE

N° 162 JUILLET/AOÛT

À NE PAS RATER !

CENTRALISEZ LA GESTION DES LOGS !

DISPONIBLE DÈS **LE 28 JUIN**
CHEZ VOTRE MARCHAND DE JOURNAUX
ET SUR : www.ed-diamond.com

VOUS VOULEZ GÉRER SANS SÉQUELLES DES BROUETTES DE DONNÉES ?

DÉCOUVREZ LE NOSQL !

par Jean-Michel Armand

Certaines problématiques sont difficilement résolubles avec des bases de données classiques de type relationnelles. Dans ces situations spéciales, il peut être utile de changer de paradigme en ce qui concerne la gestion des données et d'essayer ce que certains pourraient appeler « une solution de la dernière chance », le NoSQL.

1 Le NoSQL, qu'est-ce que c'est ?

Le NoSQL (comprendre *Not only SQL* et pas *Not SQL* comme on peut le penser) s'est construit en réaction à des problématiques difficilement résolubles avec des SGBD classiques de types relationnels.

SQL, SGBD classiques et ACID

Les SGBD classiques supportent le système de gestion de transaction **ACID** :

- *Atomic* (Atomicité) : une transaction doit pouvoir être validée ou annulée pour revenir à l'état stable précédent ;
- *Consistence* (Cohérence) : les données doivent à chaque instant être dans un état cohérent ;
- *Isolated* (Isolation) : les transactions doivent être isolées dans leur contexte. Les résultats des transactions exécutées en parallèle ne doivent pas s'entre-polluer ;
- *Durable* (Durabilité) : une fois qu'une transaction est achevée, le système est dans un nouvel état stable qui ne peut être remis en cause.

Quel est donc le problème résolu par NoSQL ? Aujourd'hui, les applications doivent pouvoir gérer de plus en plus de données, des millions, voire des milliards d'enregistrements différents doivent être stockés, modifiés, retrouvés. Il faut donc faire intervenir ce qu'on appelle la **scalabilité** : répartir sur plusieurs machines, regroupées sur un réseau, les données qui ne sont plus gérables par une seule instance. On introduit alors le théorème **CAP**, appelé également **théorème de Brewer** (voir schéma en figure 1) :

- *Consistency* (Cohérence) : tous les nœuds du système voient les mêmes données au même moment ;
- *Availability* (Disponibilité) : on peut garantir que toutes les requêtes recevront une réponse ;
- *Partition Tolerance* (Résistance au morcellement) : on peut garantir qu'à part une panne de réseau totale, le système, même découpé en plusieurs sous-systèmes, répondra correctement.

Un système distribué pourra, au mieux, gérer à chaque instant deux des trois possibilités. C'est de ce théorème de CAP qu'est né le NoSQL. Le NoSQL, c'est simplement un choix différent parmi les possibilités offertes par le théorème CAP.

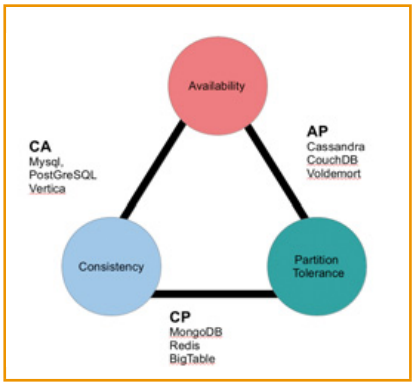


Fig. 1 : Théorème CAP

1.1 Le NoSQL, ou plutôt les NoSQL

Le NoSQL est tout simplement une façon de penser la base de données, qui n'axe pas forcément les choses sur

un sacro-saint respect de la gestion des transactions en mode ACID et qui propose d'autres possibilités de choix dans la triplette CAP. On ne parlera plus alors de système consistant, mais de système *éventuellement* consistant. De plus, en menant cette réflexion de changement de paradigme, d'autres ont suivi. Par exemple, les notions de base utilisées pour le stockage d'informations ne sont plus ni la table, ni la cellule. À la place, on verra qu'il y a plusieurs possibilités de choix d'unité logique minimale pour le stockage des informations.

1.1.1 Un peu d'histoire

Historiquement, il faut rappeler que les systèmes de bases de données non relationnels sont plus anciens que les SGBD relationnels et que donc, le mouvement NoSQL est en quelque sorte un retour aux sources. En effet, sur les *mainframes* par exemple, c'était bien des systèmes de bases de données non relationnels que l'on trouvait. Mais après cet âge d'or, les systèmes non relationnels ont été éclipsés par les systèmes relationnels. Et cela jusqu'en 1998, année où le vocable NoSQL a été utilisé pour la première fois. Dix ans plus tard, en 2009, le *meetup NoSQL* de San Francisco sera l'événement fondateur de toute une mouvance dans la théorie des bases de données. À cette occasion, plus d'une centaine de développeurs se rencontreront pour présenter leurs travaux sur différents moteurs de bases de données tels que Voldemort, Cassandra, CouchDB ou MongoDB. La grande aventure du NoSQL était sur le point de « redémarrer »...

1.1.2 Les différents types de NoSQL

On l'a déjà dit, l'une des différences entre les SGBD classiques et le NoSQL c'est le changement de paradigme concernant les unités de stockage de base, qui ne sont plus la table et la cellule. Mais là où les SGBD proposent une seule possibilité, le mouvement NoSQL propose différents paradigmes possibles. Les voici listés :

- **Système clé/valeur** : l'un des plus vieux et des plus simples systèmes de stockage de données. Chaque valeur est assignée à une clé. On peut récupérer une valeur grâce à sa clé, ou la modifier. Ce système est très souvent utilisé pour gérer par exemple des systèmes de cache. En général, c'est un système qui donne tout son potentiel lorsque les opérations de lecture sont grandement supérieures en nombre aux opérations d'écriture. On va trouver ici des systèmes tels que Redis **[1]** ou Memcached **[2]**.
- **Système orienté colonnes** : les systèmes orientés colonnes sont une simple extension des systèmes clé/valeur. Ils proposent deux niveaux de partitionnement clé/valeur. Ce système est typiquement utilisé lorsque l'on a à la fois beaucoup d'opérations de lecture ou d'écriture. On peut citer comme exemple BigTable de Google ou Cassandra **[3]** créé par Facebook.
- **Système orienté documents** : ici, on ne stocke plus que des documents. Des documents sont des objets complexes, contenant des informations structurées ou semi-structurées de plusieurs types. On va pouvoir ranger ensemble les documents dans des collections et on pourra ensuite faire des requêtes sur un ou plusieurs des éléments contenus dans les documents. On est ici totalement dans un système sans schéma. Chaque document peut en effet avoir une structure totalement unique. Les données sont stockées sous forme d'XML de JSON ou de BSON. Les bases de données orientées documents sont très souvent utilisées pour stocker les données d'applications web. On va retrouver ici CouchDB **[4]** et MongoDB **[5]**.
- **Système de graphe** : ici, on va utiliser les notions mathématiques de graphe pour stocker les données sous forme d'arêtes et de nœuds. Chaque nœud du graphe stockera différentes propriétés qui lui seront propres, ainsi que les relations qui le relient à d'autres nœuds. Les moteurs de réseaux sociaux ou de recommandations utiliseront ce type de base de données. Neo4J **[6]** et Infogrid **[7]** sont deux bons exemples de ce type de système.

Après cette très rapide présentation, nous allons tester deux systèmes NoSQL. Nous commencerons par MongoDB, base de données orientée documents, et terminerons par Redis, l'un des systèmes clé/valeur les plus connus.

2 MongoDB

Nous allons commencer par tester MongoBD, une base de données développée depuis 2010 par 10gen. Contrairement à d'autres systèmes orientés documents comme CouchDB, MongoDB ne propose pas d'API Rest pour pouvoir interagir directement avec elle. Il faudra utiliser le client fourni avec MongoDB ou l'une des librairies disponibles pour communiquer avec votre base. Le stockage de vos documents, ainsi que la communication entre MongoDB et votre application se fera en BSON (du JSON binaire). MongoDB est développé en C++ et il est publié sous licence AGPL, version 3.

2.1 Installation

2.1.1 Grâce à votre gestionnaire de paquets

Cela reste la méthode la plus facile à mettre en place. Si vous êtes sur une distribution basée sur Debian :

```
~$ sudo aptitude install mongodb
```

2.1.2 Grâce aux paquets fournis par 10gen

10gen fournit des paquets récents pour un certain nombre de distributions (Fedora, Red Hat Enterprise, CentOS, Ubuntu ou Debian). Toujours sur Debian-like, il faudra exécuter :

```
~$ sudo  echo 'deb http://downloads-distro.mongodb.org/repo/ubuntu-upstart dist 10gen' | sudo
tee /etc/apt/sources.list.d/10gen.list
~$ sudo aptitude update
~$ sudo aptitude install mongodb-10gen
```

2.2 Notions de base

En MongoDB, on stocke des documents qui prendront la forme d'un ensemble clé/valeur. Un document pourra être stocké dans une **collection**. On pourrait voir une similarité entre une collection et une table SQL classique mais, alors que dans une table chaque ligne doit avoir la même structure, les documents stockés dans des collections n'ont en aucun cas l'obligation d'avoir la même structure.

On pourra, pour plus de clarté, regrouper les collections en sous-collections en utilisant un **.** entre le nom d'une collection et de ses sous-collections. Imaginons que l'on ait une collection **blog**, on pourra donc avoir les sous-collections **blog.posts** et **blog.comments**. Au-dessus des collections, on va trouver les **databases**. Une database se contente d'être un regroupement de collections.

2.3 Convention de nommage

MongoDB met en place quelques restrictions en ce qui concerne le nommage des choses. Concernant les clés des documents :

- Ce doit être des chaînes UTF-8 ;
- Elles ne doivent pas contenir le caractère **\0** (qui est utilisé pour définir la fin de la clé) ;
- Les caractères **.** et **$** ont des significations spéciales et ne doivent pas être utilisés en dehors de cela. Il vaut mieux les considérer comme réservés ;
- Les clés qui commencent par un **_** sont considérées comme réservées ;

Les règles de nommage de collections sont :

- Une collection ne peut avoir un nom vide ;
- Le nom d'une collection ne doit pas :
    - contenir le caractère **\0** ;
    - commencer par **system** ;
    - contenir le caractère **$**.

Enfin, voici les règles concernant les databases :

- Le nom ne doit pas être vide ;
- Il ne doit pas contenir le caractère **\0**, **:**, espace, **.**, **$**, **/** ou **** ;
- Il doit toujours être en minuscules ;
- Il ne doit pas dépasser 64 octets ;
- Il ne doit pas être **admin**, **local** ou **config**.

2.4 Jouons avec MongoDB !

2.4.1 Avec le shell fourni

Comme dit plus haut, MongoDB fournit un shell. Ce shell se lance tout simplement en tapant **mongo** :

```
~$ mongo
MongoDB shell version: 2.0.4
connecting to: test
```

Vérifiez avant de lancer votre shell Mongo que votre moteur MongoDB s'exécute. En effet, le shell Mongo tente de se connecter dès que vous le lancez. Mongo est plus qu'un shell, c'est un interpréteur JavaScript avec lequel vous allez pouvoir interagir avec MongoDB. Bien que cela soit amusant de taper du code JavaScript, il existe la console Firefox pour cela. Commençons donc à jouer avec MongoDB...

Tout d'abord, il va falloir se connecter à une base. Au lancement du shell, vous allez être automatiquement connecté à la base **test**, mais on va changer cela. Imaginons que nous voulions faire des tests concernant une future application de critique de films par exemple. Nous allons nous connecter à notre base **critiques** en faisant :

```
> use critiques
switched to db critiques
```

Mongo va vous connecter à la base **critiques** (même si elle n'existait pas avant que vous ne vous y connectiez). Mais comment utiliser votre base ? Une variable très importante dans le shell Mongo est la variable **db**, qui va vous permettre d'interagir avec vos bases.

Si maintenant vous tapez **db**, vous verrez que Mongo va vous indiquer que vous êtes bien sur la base **critiques** :

```
> db
critiques
```

2.4.1.1 Création, suppression et modification de documents

Nous allons commencer par créer un document qui représentera un premier film. Par exemple :

```
> film = { "titre" : "Le hobbit : un voyage inattendu" ,
           "résumé" : "c'est l'histoire d'un hobbit qui …",
           "réalisateur" : "Peter Jackson",
           "date de sortie" : new Date("2012/12/12")
         }
```

Une fois notre document créé, il faut l'insérer dans notre base. On le fait d'une manière toute simple :

```
> db.films.insert(film)
```

Pour vérifier que notre film a bien été sauvé, on va utiliser la fonction **find()** qui sans argument renvoie la totalité d'une collection. :

```
> db.films.find()
{ "_id" : ObjectId("51a438ca5a21559a832156cb"), "titre" : "Le hobbit :
un voyage inattendu", "résumé" : "c'est l'histoire d'un hobbit qui …",
"réalisateur" : "Peter Jackson", "date de sortie" : ISODate("2012-12-
11T23:00:00Z") }
```

On voit ici que MongoDB a ajouté un attribut à notre objet, c'est la clé **_id**. Cette clé est unique par collection et permet de référencer le document. C'est en fait une sorte de clé primaire.

Si nous avions eu plusieurs films, on aurait pu ne vouloir qu'un seul film en utilisant la fonction **findOne()**.

Pour mettre à jour un document, on peut utiliser la fonction **update**. Imaginons que je veuille à la fois modifier le résumé et ajouter des commentaires à mon film. Voila comment faire :

```
> film.commentaires = []
[ ]

> db.films.update({titre : "Le hobbit : un voyage inattendu"}, film)
> db.films.find()
{ "_id" : ObjectId("51a4423e4542a4b73e56876c"), "titre" : "Le hobbit :
un voyage inattendu", "résumé" : "c'est l'histoire d'un hobbit qui …",
"réalisateur" : "Peter Jackson", "date de sortie" : ISODate("2012-12-
11T23:00:00Z"), "commentaires" : [ ] }

> film.résumé = "un nouveau résumé "
un nouveau résumé

> db.films.update({titre : "Le hobbit : un voyage inattendu"}, film)
> db.films.find()
{ "_id" : ObjectId("51a4423e4542a4b73e56876c"), "titre" : "Le hobbit :
un voyage inattendu", "résumé" : "un nouveau résumé ", "réalisateur"
: "Peter Jackson", "date de sortie" : ISODate("2012-12-11T23:00:00Z"),
"commentaires" : [ ] }
```

Et si maintenant nous voulons supprimer le film que nous avons ajouté ? Rien de plus simple :

```
> db.films.remove({titre : "Le hobbit : un voyage inattendu"})
> db.films.find()
```

Ici, on utilise la fonction **remove** qui, si on ne lui donne pas de paramètre de filtre, supprime la totalité des documents de la collection. Comme je ne voulais supprimer que le film « Le hobbit », j'ai donc donné un filtre portant sur le titre ; j'aurais pu utiliser n'importe lequel des champs de mon document. Bien entendu, si plusieurs documents valident le filtre, ils seront tous supprimés.

Pour modifier de manière plus simple les documents déjà stockés dans la base, on peut utiliser ce que l'on appelle des **modificateurs**. Par exemple, on veut ajouter une note à nos critiques de films. Pour ce faire, rien de plus simple : on va utiliser le modificateur **$set** qui, soit change la valeur d'une clé si la clé existe déjà dans le document, soit crée la nouvelle clé puis lui donne une valeur (dans les lignes suivantes, j'ai recréé le film, comme vous pouvez le constater avec l'**ObjectId** qui a changé ; je ne remets pas les lignes de création pour ne pas surcharger l'exemple) :

```
> db.films.update({ "_id" : ObjectId("51a52da65ec115cf0b8ccbea") },
{ "$set" : { "note" : 1}})
> db.films.find()
{ "_id" : ObjectId("51a52da65ec115cf0b8ccbea"), "date de sortie" :
ISODate("2012-12-11T23:00:00Z"), "note" : 1, "réalisateur" : "Peter
Jackson", "résumé" : "c'est l'histoire d'un hobbit qui …", "titre"
: "Le hobbit : un voyage inattendu" }
```

La mise à jour de mon document se fait en ligne 1. Tout d'abord, j'utilise comme critère de recherche l'id de mon document que je sais être unique. Ensuite, je lui passe le modificateur **$set** comme clé de modification et je lui donne la paire clé/valeur que je veux ajouter.

$set ne se limite pas à créer de nouvelles clés ou à modifier les valeurs de celles-ci, il peut aussi être utilisé pour changer le type d'une valeur. Voyons quelques petits exemples d'utilisation de **$set** :

```
> db.films.update({ "_id" : ObjectId("51a52da65ec115cf0b8ccbea") },
{ "$set" : { "acteurs" : "martin Freeman"}})
> db.films.update({ "_id" : ObjectId("51a52da65ec115cf0b8ccbea") },
{ "$set" : { "acteurs" : ["martin Freeman"]}})
> db.films.update({ "_id" : ObjectId("51a52da65ec115cf0b8ccbea") },
{ "$set" : { "acteurs" : ["martin Freeman", "richard armitage"]}})
> db.films.find()
{ "_id" : ObjectId("51a52da65ec115cf0b8ccbea"), "acteurs" :
[ "martin Freeman", "richard armitage" ], "date de sortie" :
ISODate("2012-12-11T23:00:00Z"), "note" : 1, "réalisateur" : "Peter
Jackson", "résumé" : "c'est l'histoire d'un hobbit qui …", "titre"
: "Le hobbit : un voyage inattendu" }
```

Je commence par ajouter un premier acteur en ajoutant une donnée **acteurs**. Mais je me trompe, j'ajoute une donnée de type chaîne de caractères au lieu d'ajouter une

liste ; donc, en ligne 2, je modifie le type de la valeur de ma clé **acteurs** pour que ce soit une liste. Et enfin, j'ajoute un nouvel acteur à ma liste.

Il existe le pendant de **$set**, **$unset**, qui permet de supprimer une clé et sa valeur :

```
> db.films.update({ "_id" : ObjectId("51a
52da65ec115cf0b8ccbea") }, { "$unset" : {
"acteurs" : 1}})
> db.films.find()
{ "_id" : ObjectId("51a52da65ec115cf0b8ccb
ea"), "date de sortie" : ISODate("2012-12-
11T23:00:00Z"), "note" : 1, "réalisateur"
: "Peter Jackson", "résumé" : "c'est
l'histoire d'un hobbit qui …", "titre" :
"Le hobbit : un voyage inattendu" }
```

$inc permet d'incrémenter et de décrémenter la valeur d'une clé. Pour augmenter la note du film de 3, il suffira de faire :

```
> db.films.update({ "_id" : ObjectId("51
a52da65ec115cf0b8ccbea") }, { "$inc" : {
"note" : 3}})
> db.films.find()
{ "_id" : ObjectId("51a52da65ec115cf0b8ccb
ea"), "date de sortie" : ISODate("2012-12-
11T23:00:00Z"), "note" : 4, "réalisateur"
: "Peter Jackson", "résumé" : "c'est
l'histoire d'un hobbit qui …", "titre" :
"Le hobbit : un voyage inattendu" }
```

Pour décrémenter une valeur, il suffit de donner une valeur négative à l'incrément dans l'appel à **update**. Nous allons terminer ce bref aperçu des modificateurs utilisables avec **update** en parlant des modificateurs travaillant sur les tableaux. En effet, comme vous l'avez vu plus haut quand j'ai voulu modifier les acteurs du film, j'ai utilisé **$set**, mais j'ai dû du coup redonner la totalité des valeurs de mon tableau, ce qui n'est pas très intéressant…

À la place de **$set**, j'aurais pu utiliser le modificateur **$push**, qui permet d'ajouter un élément en fin de tableau. Le problème de **$push** est qu'il ne vérifie pas si l'élément que l'on veut ajouter est déjà présent. Cela peut être embêtant lorsque l'on ne veut pas avoir de doublons. Il y alors deux solutions. La première est d'utiliser un modificateur conditionnel dans la requête de sélection du document, comme par exemple le modificateur **$ne**, qui est un modificateur de non présence. La seconde est d'utiliser **$addToSet** à la place de **$push**, **$addToSet** gérant les doublons.

```
> db.films.update({ "_id" : ObjectId("51a52da65ec115cf0b8ccbea") }, { "$push" : { "acteurs" :
"Ian McKellen"}})
> db.films.find()
{ "_id" : ObjectId("51a52da65ec115cf0b8ccbea"), "acteurs" : [ "martin Freeman", "richard
armitage", "Ian McKellen" ], "date de sortie" : ISODate("2012-12-11T23:00:00Z"), "note" : 4,
"réalisateur" : "Peter Jackson", "résumé" : "c'est l'histoire d'un hobbit qui …", "titre" :
"Le hobbit : un voyage inattendu" }

> db.films.update({ "_id" : ObjectId("51a52da65ec115cf0b8ccbea"), "acteurs" : {"$ne" : "Ian
McKellen"} }, { "$push" : { "acteurs" : "Ian McKellen"}})
> db.films.find()
{ "_id" : ObjectId("51a52da65ec115cf0b8ccbea"), "acteurs" : [ "martin Freeman", "richard
armitage", "Ian McKellen" ], "date de sortie" : ISODate("2012-12-11T23:00:00Z"), "note" : 4,
"réalisateur" : "Peter Jackson", "résumé" : "c'est l'histoire d'un hobbit qui …", "titre" :
"Le hobbit : un voyage inattendu" }

> db.films.update({ "_id" : ObjectId("51a52da65ec115cf0b8ccbea") }, { "$addToSet" : {
"acteurs" : "Ken Stott"}})
> db.films.find()
{ "_id" : ObjectId("51a52da65ec115cf0b8ccbea"), "acteurs" : [ "martin Freeman", "richard
armitage", "Ian McKellen", "Ken Stott" ], "date de sortie" : ISODate("2012-12-11T23:00:00Z"),
"note" : 4, "réalisateur" : "Peter Jackson", "résumé" : "c'est l'histoire d'un hobbit qui …",
"titre" : "Le hobbit : un voyage inattendu" }

> db.films.update({ "_id" : ObjectId("51a52da65ec115cf0b8ccbea") }, { "$addToSet" : {
"acteurs" : "Ken Stott"}})
> db.films.find()
{ "_id" : ObjectId("51a52da65ec115cf0b8ccbea"), "acteurs" : [ "martin Freeman", "richard
armitage", "Ian McKellen", "Ken Stott" ], "date de sortie" : ISODate("2012-12-11T23:00:00Z"),
"note" : 4, "réalisateur" : "Peter Jackson", "résumé" : "c'est l'histoire d'un hobbit qui …",
"titre" : "Le hobbit : un voyage inattendu" }
```

L'autre avantage de **$addToSet** est de pouvoir le combiner avec le modificateur **$each**. Imaginons que je veuille ajouter plusieurs acteurs en une seule fois, je vais pouvoir faire la chose suivante :

```
> db.films.update({ "_id" : ObjectId("51a52da65ec115cf0b8ccbea") }, { "$addToSet" : {
"acteurs" : { "$each" : ["Ken Stott", "James Nesbitt", "Stephen Hunter"]}}})
> db.films.find()
{ "_id" : ObjectId("51a52da65ec115cf0b8ccbea"), "acteurs" : [ "martin Freeman", "richard
armitage", "Ian McKellen", "Ken Stott", "James Nesbitt", "Stephen Hunter" ], "date de sortie"
: ISODate("2012-12-11T23:00:00Z"), "note" : 4, "réalisateur" : "Peter Jackson", "résumé" :
"c'est l'histoire d'un hobbit qui …", "titre" : "Le hobbit : un voyage inattendu" }
```

Bien entendu, on peut vouloir supprimer des éléments du tableau. Pour cela, il existe deux modificateurs : soit **$pop**, soit **$pull**. **$pop** prend en argument **1** ou **-1** selon que l'on veuille enlever un élément à partir du début ou de la fin ; quant à **$pull**, il supprimera les éléments qui valideront le critère qu'on lui aura passé en paramètre :

```
> db.films.update({ "_id" : ObjectId("51a52da65ec115cf0b8ccbea") }, { "$pop" : {"acteurs" :
1 }})
> db.films.update({ "_id" : ObjectId("51a52da65ec115cf0b8ccbea") }, { "$pull" : {"acteurs" :
"richard armitage" }})
> db.films.find()
{ "_id" : ObjectId("51a52da65ec115cf0b8ccbea"), "acteurs" : [ "martin Freeman", "Ian
McKellen", "Ken Stott", "James Nesbitt" ], "commentaires" : [ ], "date de sortie" :
ISODate("2012-12-11T23:00:00Z"), "réalisateur" : "Peter Jackson", "résumé" : "un nouveau
résumé ", "titre" : "Le hobbit : un voyage inattendu" }
```

Enfin, on peut modifier les valeurs stockées dans un tableau en utilisant les positions dans le tableau, ou en recherchant les items du tableau qui valident une requête :

```
> db.films.update({ "_id" : ObjectId("51a52da65ec115cf0b8ccbea") }, {
"$set" : {"acteurs.0" : "Martin Freeman" }})
> db.films.update({ "_id" : ObjectId("51a52da65ec115cf0b8ccbea"),
"acteurs" : "James Nesbitt" }, { "$set" : {"acteurs.$" : "William
Kircher" }})
> db.films.find()
{ "_id" : ObjectId("51a52da65ec115cf0b8ccbea"), "acteurs" : [
"Martin Freeman", "Ian McKellen", "Ken Stott", "William Kircher"
], "commentaires" : [ ], "date de sortie" : ISODate("2012-12-
11T23:00:00Z"), "réalisateur" : "Peter Jackson", "résumé" : "un
nouveau résumé ", "titre" : "Le hobbit : un voyage inattendu" }
```

Ici, j'indique explicitement que je veux modifier le premier item du tableau qui a donc l'index **0**. Par contre, avec le **acteurs.$**, j'indique que je veux modifier les items du tableau **acteurs** qui seront remontés par ma requête de filtrage qui contient un critère se basant sur **acteurs**. Lorsque vous utilisez l'opérateur de positionnement indéfini **$**, il faut impérativement que vos requêtes de sélection possèdent un critère concernant le tableau que vous voulez modifier !

Pour en finir avec la modification de documents, quelques petites choses encore. Sachez qu'en passant **true** comme dernier argument d'**update**, vous pourrez créer un document si **update** ne le trouve pas.

```
> db.films.update({ "titre" : "ET"}, {"$set" : { "résumé ": "ET veut
rentrer chez lui" }}, true)
> db.films.find()
{ "_id" : ObjectId("51a52da65ec115cf0b8ccbea"), "acteurs" : [
"Martin Freeman", "Ian McKellen", "Ken Stott", "William Kircher"
], "commentaires" : [ ], "date de sortie" : ISODate("2012-12-
11T23:00:00Z"), "réalisateur" : "Peter Jackson", "résumé" : "un
nouveau résumé ", "titre" : "Le hobbit : un voyage inattendu" }
{ "_id" : ObjectId("51a532661c45b55c67176266"), "résumé " : "ET veut
rentrer chez lui", "titre" : "ET" }
```

Enfin, il existe une fonction d'aide **save** qui prend en paramètre un document et qui, soit l'ajoutera s'il n'existe pas déjà dans la collection, soit le mettra à jour. Cela permet d'éviter d'avoir à faire des updates avec comme unique critère une recherche sur l'**_id**.

2.4.1.2 Faire des requêtes

Maintenant que nous avons vu comment créer, modifier et supprimer des documents, il est temps de voir comment on va pouvoir faire des requêtes sur notre base de données. Les fonctions qui vont vous servir à effectuer des requêtes seront **find()** et **findOne()**, que nous avons déjà vues auparavant. Vous allez pouvoir à la fois faire des requêtes sur des critères conditionnels (des documents qui contiennent ou ne contiennent pas certaines clés, dont les valeurs de certaines clés sont bien précises), mais aussi des requêtes complexes intégrant des fonctions que vous écrirez directement en JavaScript. Enfin, bien entendu, vous pourrez limiter le nombre d'éléments remontés par vos requêtes ou les classer.

MongoDB et le fire and forget

Par défaut, MongoDB fonctionne en mode « fire and forget ». Lorsque vous lancez une commande en direction de votre moteur de base de données (comme un **update** ou un **insert**), le client MongoDB n'attend pas que la base de données lui confirme la bonne exécution de l'opération. Cela permet d'augmenter d'une manière très impressionnante la vitesse des choses, mais d'un autre côté, cela diminue la confiance que l'on peut avoir dans la base de données. Pour forcer Mongo à attendre la réponse du moteur de base de données et donc, être sûr que les choses se soient bien passées ou au contraire ont échoué, il faut ajouter l'argument **safe=true** lorsque vous effectuez des commandes. Par exemple, **db.films.insert(film, safe=true)**.

Testons différentes requêtes (d'autres films ont été ajoutés à la base) :

```
> db.films.find({"titre" : "D.A.R.Y.L" })
{ "_id" : ObjectId("51a5f6cf1c45b55c6717626a"), "acteurs" : [
"Barret Oliver", "Mary Beth Hurt" ], "résumé" : "Daryl, ...",
"titre" : "D.A.R.Y.L" }
```

Ici, on ne fait qu'un **find()** simple, comme on en a fait un certain nombre jusqu'ici, sur une valeur d'une clé du document.

```
> db.films.find({"titre" : "D.A.R.Y.L" }, {"titre" : 1, "acteurs"
: 1 })
{ "_id" : ObjectId("51a5f6cf1c45b55c6717626a"), "acteurs" : [
"Barret Oliver", "Mary Beth Hurt" ], "titre" : "D.A.R.Y.L" }
```

On fait à peu près la même chose que précédemment, mais on ajoute un dictionnaire permettant de choisir quelles sont les clés que l'on veut récupérer. Ici, on veut donc simplement avoir le titre du film et les acteurs.

```
> db.films.find({"titre" : "D.A.R.Y.L" }, {"titre" : 1, "acteurs" :
1, "_id" : 0 })
{ "acteurs" : [ "Barret Oliver", "Mary Beth Hurt" ], "titre" :
"D.A.R.Y.L" }
```

Ici, c'est à peu près la même chose, sauf que l'on ajoute explicitement le fait que l'on ne veut pas le champ **_id**.

```
> db.films.find({"date de sortie": {"$gte": new Date("01/01/1988")
}})
{ "_id" : ObjectId("51a52da65ec115cf0b8ccbea"), "acteurs" : [
"Martin Freeman", ... ], "commentaires" : [ ], "date de sortie" :
ISODate("2012-12-11T23:00:00Z"), "réalisateur" : "Peter Jackson",
"résumé" : "un nouveau résumé ", "titre" : "Le hobbit : un voyage
inattendu" }
{ "_id" : ObjectId("51a5efe31c45b55c67176267"), "acteurs" : [ "Bruce
Willis" ], "date de sortie" : ISODate("1988-09-20T22:00:00Z"),
"résumé" : "John McClane...", "titre" : "Die Hard 1" }
```

On commence à faire de vrais tests sur nos requêtes. On utilise en effet **$gte** pour ne récupérer que les films dont la date de sortie est postérieure au 1er janvier 1988.

```
> db.films.find({"note": {"$lte": 20  }})
{ "_id" : ObjectId("51a5efe31c45b55c67176267"), "acteurs" : [ "Bruce
Willis" ], "date de sortie" : ISODate("1988-09-20T22:00:00Z"),
"note" : 12, "résumé" : "John McClane...", "titre" : "Die Hard 1" }
```

C'est un peu la même chose, mais pour les notes inférieures à 20 (avec **$lte**). **$lte** et **$gte** ne sont pas les seules conditions utilisables, vous pourrez utilisez aussi **$mod**, **$lt** , **$gt** et **$ne**. **$ne** fonctionne sur tous les types de données et correspond à un « n'est pas égal à ». Enfin, **$not** est une condition qui s'applique au-dessus d'une condition pour en inverser le sens. Par exemple, pour avoir les notes supérieures à 20, j'aurais pu écrire : **{ "$not" : {"note": {"$lte": 20 }}}**.

```
> db.films.find({"note": {"$in": [12, 22]  }})
{ "_id" : ObjectId("51a5efe31c45b55c67176267"), "acteurs"
: [ "Bruce Willis" ], "date de sortie" : ISODate("1988-09-
20T22:00:00Z"), "note" : 12, "résumé" : "John McClane...", "titre"
: "Die Hard 1" }
{ "_id" : ObjectId("51a5f4be1c45b55c67176268"), "acteurs" : [
"Barret Oliver", "Mary Beth Hurt" ], "note" : 22, "résumé" :
"Daryl,...", "titre" : "D.A.RY.L" }
```

On utilise le modificateur **$in** pour chercher les films dont la note est contenue dans une liste passée en argument de **$in**.

```
> db.films.find({"date de sortie": null })
{ "_id" : ObjectId("51a532661c45b55c67176266"), "résumé " :
"ET...", "titre" : "ET" }
...
{ "_id" : ObjectId("51a5f4be1c45b55c67176268"), "acteurs" : [
"Barret Oliver", "Mary Beth Hurt" ], "note" : 22, "résumé" :
"Daryl, ...", "titre" : "D.A.RY.L" }
```

Je veux chercher tous les films qui n'ont pas de date de sortie : j'utilise pour cela **null** comme critère de recherche.

```
> db.films.find({"titre": /HoBbit/ })
> db.films.find({"titre": /HoBbit/i })
{ "_id" : ObjectId("51a52da65ec115cf0b8ccbea"), "acteurs" : [
"Martin Freeman", ... ], "commentaires" : [ ], "date de sortie" :
ISODate("2012-12-11T23:00:00Z"), "réalisateur" : "Peter Jackson",
"résumé" : "un nouveau résumé ", "titre" : "Le hobbit : un voyage
inattendu" }
```

J'utilise des expressions régulières pour trouver tous les films contenant « hobbit ». La seule différence est l'utilisation du **i** dans la seconde ligne, qui permet de ne pas gérer la casse dans la recherche.

```
> db.films.find({"acteurs": /bruce/i })
{ "_id" : ObjectId("51a5efe31c45b55c67176267"), "acteurs"
: [ "Bruce Willis" ], "date de sortie" : ISODate("1988-09-
20T22:00:00Z"), "note" : 12, "résumé" : "John McClane...", "titre"
: "Die Hard 1" }
```

Nous avons encore ici une expression régulière, mais pour chercher une chaîne parmi un tableau de chaînes.

```
> db.films.find({"acteurs": {"$all": ["Martin Freeman", "Ian McKellen"]
}})
{ "_id" : ObjectId("51a52da65ec115cf0b8ccbea"), "acteurs" : [ "Martin
Freeman", ... ], "commentaires" : [ ], "date de sortie" : ISODate("2012-
12-11T23:00:00Z"), "réalisateur" : "Peter Jackson", "résumé" : "un
nouveau résumé ", "titre" : "Le hobbit : un voyage inattendu" }
```

J'utilise le modificateur **$all**, qui me permet de spécifier que je veux trouver les films dans lesquels ont joué tous les acteurs donnés en paramètres de **$all**. Si je n'avais pas utilisé **$all** mais juste **{"acteurs": ["Martin Freeman", "Ian McKellen"] }**, je n'aurais eu comme résultat que les films ayant comme acteurs uniquement Martin Freeman et Ian McKellen et donc, je n'aurais pas eu « Le hobbit » comme résultat.

```
> db.films.find({ "$where" : "function() { return this.note > 12;}"})
{ "_id" : ObjectId("51a5f4be1c45b55c67176268"), "acteurs" : [ "Barret
Oliver", "Mary Beth Hurt" ], "note" : 22, "résumé" : "Daryl,...",
"titre" : "D.A.RY.L" }
{ "_id" : ObjectId("51a5f6441c45b55c67176269"), "acteurs" : [ "Will
Wheaton", "Barbara Bosson" ], "date de sortie" : ISODate("1983-12-
31T23:00:00Z"), "note" : 25, "résumé" : "...", "titre" : "The Last
Starfighter" }
```

On atteint ici toute la puissance du système de requêtes de MongoDB. Celui-ci permet en effet, grâce au modificateur **$where**, d'utiliser une fonction JavaScript comme critère de choix.

```
> db.films.find({ "$where" : "function() { return this.note > 12;}"}).
limit(1)
{ "_id" : ObjectId("51a5f4be1c45b55c67176268"), "acteurs" : [ "Barret
Oliver", "Mary Beth Hurt" ], "note" : 22, "résumé" : "Daryl,...",
"titre" : "D.A.RY.L" }
```

Enfin, je limite ici le nombre de retours à 1 avec la fonction **limit()**. De manière presque antinomique à **limit()**, vous allez pouvoir utiliser **skip()**, qui supprimera les **n** premiers résultats de votre recherche, **n** étant le paramètre que vous aurez donné à **skip()**.

Enfin, vous pouvez utiliser **sort()**, qui prend en argument un dictionnaire clé/valeur qui permet de trier en fonction des clés. Les valeurs pour chaque clé peuvent être **1** ou **-1**, selon que vous vouliez trier de manière croissante ou décroissante.

2.4.1.3 Faire des index

Savoir faire des requêtes c'est très bien. Mais quand on a énormément de données - et c'est quasiment toujours le cas si une architecture NoSQL a été choisie -, le temps d'exécution des requêtes a tendance à augmenter très rapidement. Les index sont là pour faire en sorte que les temps d'exécution des requêtes de recherche restent raisonnables. Par contre, ce gain de rapidité ne se fait pas sans contrepartie. En effet, un index implique de travailler sur des données qui seront présentes dans tous les documents d'une collection. On dégrade donc un

peu notre paradigme « schema-free » en ajoutant des contraintes ; par exemple, que tous les documents d'une collection auront une clé **date** sur laquelle on va construire un index.

Pour créer un index en MongoDB, c'est relativement simple. Il suffit d'utiliser la fonction **ensureIndex()** :

```
> db.films.ensureIndex({"titre" : 1 })
> db.films.ensureIndex({"note" : 1 })
> db.films.ensureIndex({"titre" : 1 ,
"note" : -1})
```

Ici, je crée trois index. On peut créer, comme en dernière ligne, des index sur plusieurs clés à la fois. Les valeurs correspondent au sens dans lequel on va indexer. Cette valeur n'a pas d'importance sur des index à une clé comme ceux que j'ai créés dans les deux premières lignes, par contre il est important d'y réfléchir lorsque l'on crée des index sur plusieurs clés.

Trop est l'ennemi de bien

Les index augmentent la rapidité des requêtes. On pourrait donc imaginer faire des index sur toutes les clés d'une collection en imaginant que nos requêtes vont alors devenir fulgurantes. Cela ne sera pas vraiment le cas. Par contre, cela va grandement ralentir vos insertions et multiplier de façon importante la place consommée par votre base. Plutôt que de créer des index à la tonne dont la plupart seront quasi inutiles, il vaut bien mieux que vous réfléchissiez aux requêtes que vous allez faire de manière récurrente, et que vous limitiez vos index à ceux qui vont vraiment réduire le temps d'exécution de ces requêtes-là.

Vous allez également pouvoir créer des index uniques, qui vous permettront d'affirmer que dans toute la collection, il n'y aura qu'un seul document avec cet index précis. Pour cela, il vous suffira d'ajouter le paramètre **{ "unique" : true }** lors de votre appel à **ensureIndex()**.

2.4.1.4 Faire des agrégats

MongoDB propose un certain nombre d'agrégats utilisables. Nous allons commencer par voir les plus simples. Le dernier, le roi des agrégats serais-je tenté de dire, le **MapReduce**, fera l'objet d'une partie spéciale.

```
> db.films.count()
6
> db.films.distinct('note')
[ null, 12, 22, 25 ]
> db.films.distinct('note', {'note' : { $gt : 12} } )
```

Comme on le voit, on peut compter le nombre de documents présents dans une collection. **count()** ne prend pas de paramètre de requête ; en effet, il suffira de faire un **find(<query>).count()** si on veut compter une sous-partie des documents de sa collection. On peut également demander à MongoDB de nous donner les valeurs distinctes pour une clé dans tous les documents de la collection. On remarquera que **distinct()** permet d'ajouter des conditions qui vont filtrer les documents de la collection. Enfin, on peut également grouper les choses comme en SQL, en utilisant la fonction **db.collections.group()**.

2.4.1.5 Le MapReduce

Le MapReduce est en fait l'un des intérêts des bases de données NoSQL. Le principe du MapReduce est simple : il consiste à partitionner les données afin de pouvoir les traiter en parallèle. Une fois les partitions indépendantes traitées, on les combinera pour avoir le résultat final.

Plus précisément, le Map concerne la partie découpage et redistribution des partitions indépendantes de l'algorithme. Une fois que les données ont été découpées en partitions indépendantes et que chaque petite partie a été traitée, c'est au tour du Reduce d'agir. Le Reduce est en fait l'exact opposé du Map : il va récupérer chacun des bouts de résultats et les fusionner en un seul résultat global.

Bien entendu, Reduce doit être capable de rassembler les données d'une manière sensée. C'est pour cela que lors du découpage en partitions de données, Map découpe les choses suivant un couple clé/valeur. C'est cette clé qui permettra à Reduce de faire son travail de fusion des résultats partiels.

La fonction **map** est donc une fonction qui doit prendre la collection et la découper en sous-ensembles, chaque sous-ensemble étant de la forme clé/valeur.

La fonction **reduce** prendra elle deux paramètres : une clé et une collection d'objets associée à cette clé. Bien entendu, la collection d'objets devra être de la même forme que la valeur qui aura été émise par **map**. Une chose est importante à ne pas oublier : la fonction **reduce** peut être appelée plusieurs fois si nécessaire, avec potentiellement en paramètres des choses qu'elle aura générée elle-même.

Pour expliciter les choses, on va utiliser MapReduce pour calculer le nombre de fois qu'un acteur apparaît dans les films stockés dans notre base. On en profitera pour afficher les films en question. Pour avoir des données un peu plus claires, on va utiliser une nouvelle collection avec des données de test :

```
> db.f2.update({ "titre" : "F1"}, {"$set" : { "acteurs" : ["a1", "a2", "a3"]}}, true)
> db.f2.update({ "titre" : "F2"}, {"$set" : { "acteurs" : ["a1", "a4", "a5"]}}, true)
> db.f2.update({ "titre" : "F3"}, {"$set" : { "acteurs" : ["a1", "a6", "a7"]}}, true)
> db.f2.update({ "titre" : "F4"}, {"$set" : { "acteurs" : ["a2", "a3", "a8"]}}, true)
> db.f2.update({ "titre" : "F5"}, {"$set" : { "acteurs" : ["a4", "a3", "a1"]}}, true)
> db.f2.update({ "titre" : "F6"}, {"$set" : { "acteurs" : ["a4", "a3", "a1"]}}, true)
```

On va maintenant écrire la fonction **map** et la fonction **reduce** :

```
> map = function() {
   for ( var i in this.acteurs) {
     emit (this.acteurs[i], { "films" : [this.titre], "nb" : 1 });
    }
};

> reduce = function(key, emits) {
    var total = { "films" : [], nb : 0}
    for ( var i in emits) {
        emits[i].films.forEach(function(film) {
            total.films.push(film);
        })
        total.nb += emits[i].nb;
    }
    return total;
};
```

La fonction **map** se contente, pour chaque item de la collection, de parcourir les acteurs et de découper les choses. On utilise comme clé le nom de l'acteur. Les données envoyées pour **reduce** se composent d'un titre de film et du chiffre 1, indiquant ainsi que l'acteur dont le nom est en clé est présent dans le film.

La fonction **reduce**, elle, s'occupe de récupérer les données partitionnées et additionne petit à petit les présences des acteurs dans les films. Il ne nous reste plus qu'à lancer le **mapReduce** :

```
> db.f2.mapReduce(map, reduce,{out:'test1'})
{
  "result" : "test1",
  "timeMillis" : 7,
  "counts" : {
    "input" : 6,
    "emit" : 18,
    "reduce" : 4,
    "output" : 8
  },
  "ok" : 1,
}

> db.test1.find()
{ "_id" : "a1", "value" : { "films" : [ "F1", "F2", "F3", "F5", "F6" ], "nb"
: 5 } }
{ "_id" : "a2", "value" : { "films" : [ "F1", "F4" ], "nb" : 2 } }
{ "_id" : "a3", "value" : { "films" : [ "F1", "F4", "F5", "F6" ], "nb" : 4 } }
{ "_id" : "a4", "value" : { "films" : [ "F2", "F5", "F6" ], "nb" : 3 } }
{ "_id" : "a5", "value" : { "films" : [ "F2" ], "nb" : 1 } }
{ "_id" : "a6", "value" : { "films" : [ "F3" ], "nb" : 1 } }
{ "_id" : "a7", "value" : { "films" : [ "F3" ], "nb" : 1 } }
{ "_id" : "a8", "value" : { "films" : [ "F4" ], "nb" : 1 } }
```

Les trois paramètres obligatoires sont la fonction **map**, la fonction **reduce** et le paramètre **out**, qui indique où l'on doit stocker les données résultantes. On peut utiliser d'autres paramètres comme le paramètre **query**, qui permettra de filtrer les documents de la collection pour n'en sélectionner qu'une sous-partie avant d'appliquer le **mapReduce**. Vous pourrez également utiliser **limit**, qui vous permettra de donner une borne maximale au nombre de documents qui seront générés dans la collection résultante. MapReduce est un outil très puissant. Son seul inconvénient est qu'il est assez lent. Et plus vous aurez de données à traiter, plus il sera intéressant, mais plus il sera lent...

2.4.2 Dans d'autres langages

Jusqu'à présent, nous n'avons utilisé MongoDB qu'à travers son shell et donc intégralement en JavaScript. Heureusement pour nous, il existe un nombre important de drivers qui permettent de communiquer avec MongoDB à partir d'autres langages. La page des drivers de MongoDB **[8]** vous listera la totalité des langages utilisables, parmi lesquels Python, PHP, Ruby, Java...

Essayons avec du Python. Tout d'abord, il faut installer le driver Python. Si vous savez utilisé les virtualenvs, il vous suffira de taper :

```
(mongodb)~$ pip install python-pymongo
```

Sinon, il faudra passer par un **sudo** :

```
~$ sudo pip install python-pymongo
```

Ensuite, il n'y a plus qu'à tester :

```
>>> from pymongo import MongoClient
>>> client = MongoClient()
>>> db = client.critiques
>>> films = db.films
>>> films.find_one()
{u'date de sortie': datetime.datetime(2012, 12, 11, 23, 0), u'acteurs':
[u'Martin Freeman', u'Ian McKellen', u'Ken Stott', u'William Kircher'],
u'r\xe9alisateur': u'Peter Jackson', u'titre': u'Le hobbit\xa0: un
voyage inattendu', u'commentaires': [], u'_id': ObjectId('51a52da65ec115
cf0b8ccbea'), u'r\xe9sum\xe9': u'un nouveau r\xe9sum\xe9 '}
```

On va commencer par se connecter à la base de données, puis récupérer notre base et notre collection. Je teste ensuite le **find_one()** :

```
>>> films.update({ "titre" : "Die Hard 3"}, {"$set" : { "résumé" : "John McClane
est je ne sais plus ou..." , "note" : 8 , "acteurs" : ["Bruce Willis"]}}, True)
{u'ok': 1.0, u'upserted': ObjectId('51a672e11c45b55c67176272'), u'err': None,
u'connectionId': 8, u'n': 1, u'updatedExisting': False}
```

L'**update** permet de créer un nouveau film.

```
>>> films.find()
<pymongo.cursor.Cursor object at 0xb6e4dcec>
>>> for f in films.find(): f
```

Pour récupérer l'intégralité de tous les films, par contre, ce n'est pas aussi simple que dans le shell. Le **find()** nous renvoie en effet un objet **Cursor**, qu'il faudra parcourir comme je le fais par la suite. Comme vous le voyez, l'API est quasiment la même et elle est tout aussi complète que dans le shell MongoDB. Vous pourrez donc sans souci utiliser MapReduce.

3 Redis

La première version de Redis (pour REmote DIctionary Server) date de 2009. Redis est développé en C, sous licence BSD. Redis est un système clé/valeur qui vise à stocker des données simples (chaînes de caractères, compteurs numériques, tableaux associatifs sous forme de dictionnaires, listes, ensembles et ensembles ordonnés). L'approche de Redis est une approche assez intéressante. En effet, bien qu'étant conçu pour être totalement en RAM, ce qui lui vaut d'avoir d'excellentes performances, il lui est aussi possible de stocker les données qu'il gére sur le disque. Enfin, c'est une base de données qui gére la réplication et la répartition de charge via un modéle de type maître/esclave.

3.1 Installation

Vous pouvez installer Redis tout simplement, grâce à votre gestionnaire de paquets :

```
~$ sudo aptitude install redis-server
```

Vous pouvez sinon le télécharger et le compiler directement :

```
~$ wget http://redis.googlecode.com/files/redis-2.6.13.tar.gz
~$ tar xzf redis-2.6.13.tar.gz
~$ cd redis-2.6.13
~$ make
```

Une fois le **make** fini, vous aurez un exécutable dans **src** : **redis-server**.

Et si jamais vous vous sentez d'humeur aventureuse, vous pouvez même récupérer le code directement sur GitHub **[9]**.

3.2 Jouons avec Redis !

3.2.1 Avec l'interface en ligne de commandes fournie

Redis s'installe avec une interface en ligne de commandes qui permet d'interagir directement avec lui. Pour la lancer, il suffit de taper la commande suivante :

```
~$ redis-cli
```

On peut alors jouer avec Redis directement. Par exemple :

```
redis 127.0.0.1:6379> EXISTS test
(integer) 1
redis 127.0.0.1:6379> EXISTS KeyLinuxMag
(integer) 0
redis 127.0.0.1:6379> APPEND KeyLinuxMag "Bonjour "
(integer) 8
redis 127.0.0.1:6379> APPEND KeyLinuxMag "tout le monde"
(integer) 21
redis 127.0.0.1:6379> GET KeyLinuxMag
"Bonjour tout le monde
```

Ici, je commence par vérifier si la clé **test** existe, ce qui est le cas, je l'avais en effet créée précédemment. Puis, je vérifie l'existence de la clé **KeyLinuxMag** et comme elle n'existe pas, je lui assigne une valeur que je modifie avant de la récupérer.

On peut également jouer avec les listes :

```
redis 127.0.0.1:6379> RPUSH List "Bla"
(integer) 1
redis 127.0.0.1:6379> RPUSH List "Bli"
(integer) 2
redis 127.0.0.1:6379> RPUSH List "Blio"
(integer) 3
redis 127.0.0.1:6379> LRANGE List
(error) ERR wrong number of arguments for 'lrange' command
redis 127.0.0.1:6379> LRANGE List 0 -1
1) "Bla"
2) "Bli"
3) "Blio"
```

On peut également définir des durées de vie pour les clés, ce qui permet de mettre en place très facilement des systèmes de cache. On pourra faire par exemple :

```
redis 127.0.0.1:6379> SET kexpire "Test expiration"
OK
```

Je déclare ma clé en lui donnant une valeur.

```
redis 127.0.0.1:6379> EXPIRE kexpire 10
(integer) 1
```

Je lui définis une durée de vie en secondes.

```
redis 127.0.0.1:6379> TTL kexpire
(integer) 5
redis 127.0.0.1:6379> TTL kexpire
(integer) 2
redis 127.0.0.1:6379> TTL kexpire
(integer) -1
```

Je vérifie que la vie de ma clé diminue.

```
redis 127.0.0.1:6379> GET kexpire
(nil)
```

Je tente de récupérer la valeur de ma clé qui n'existe plus. On pourra, si nécessaire, rendre à nouveau une clé persistante avec le mot-clé **PERSIST**.

3.2.1.1 Pub/Sub

Pub/Sub, ou *Publish/Subscribe*, est un mécanisme de publication/souscription de messages, qui pose comme hypothèse de départ que les émetteurs ne savent pas a priori quels destinataires vont recevoir leurs messages. En effet, les destinataires s'enregistrent sur des *channels*. Et les émetteurs se contentent de pousser les messages dans lesdits channels. Pour tester l'implémentation Pub/Sub de Redis, il va nous falloir lancer deux consoles **redis-cli**.

Dans la première, nous allons taper :

```
redis 127.0.0.1:6379> SUBSCRIBE channel1
Reading messages... (press Ctrl-C to quit)
1) "subscribe"
2) "channel1"
3) (integer) 1
```

Ici, nous avons perdu la main et notre **redis-cli** attend l'arrivée d'un message.

Dans une deuxième **redis-cli** nous allons alors taper :

```
redis 127.0.0.1:6379> PUBLISH channel1 "Bonjour"
(integer) 1
```

Ceci aura pour effet de faire apparaître les messages suivants dans notre premier **redis-cli** :

```
1) "message"
2) "channel1"
3) "Bonjour"
```

3.2.2 Dans d'autres langages

Là encore, Redis peut s'interfacer avec de multiples langages de programmation. Nous allons à nouveau tester la chose avec Python. Il existe plusieurs modules Python pour discuter avec un serveur Redis. J'ai décidé d'utiliser le module **python-redis** fourni dans les paquets de ma distribution :

```
~$ sudo aptitude install python-redis
```

Une fois cela fait, il n'y a plus qu'à lancer votre console Python préférée et lancer quelques tests :

```
>>> import redis
>>> c = redis.Redis()
>>> c.set('Key',"Ma valeur de test")
True
>>> c.get('Key')
'Ma valeur de test'
```

Comme vous le voyez, la syntaxe reste très similaire à l'API de la **redis-cli**. Je donne une valeur à une clé, valeur que je récupère ensuite.

```
>>> c.rpush('liste', 'val1')
1L
>>> c.rpush('liste', 'val2')
2L
>>> c.rpush('liste', 'val3')
3L
>>> c.lrange('liste',0, -1)
['val1', 'val2', 'val3']
```

Je remplis une liste que j'affiche.

```
>>> c.sadd('set', 1)
True
>>> c.sadd('set', 2)
True
>>> c.sadd('set', 2)
False
>>> c.smembers('set')
set(['1', '2'])
```

Je remplis un ensemble dont je demande l'affichage.

```
>>> c.hset('dico', 'cle1', 'Val1')
1L
>>> c.hset('dico', 'cle2', 'Val2')
1L
>>> c.hget('dico', 'cle2')
'Val2'
```

Pour finir, je mets des valeurs dans un dictionnaire avant de récupérer la valeur de l'une de ses clés.

Comme vous pouvez le voir, utiliser Redis est très simple. Et c'est justement cette simplicité qui fait sa grande force et le fait qu'il soit énormément utilisé pour mettre en place des systèmes de cache ou de queue d'exécution.

Conclusion

Cet article n'est qu'une légère introduction au domaine du NoSQL qui foisonne de vie, de trolls et de possibilités. Bien que le NoSQL ne soit certainement pas la réponse à toutes les problématiques de gestion de base de données, je ne saurais que trop vous conseiller de ne pas oublier de penser à ces outils lorsque vous réfléchirez à votre projet utilisant un système de stockage de données. Et si vous vouliez une seule preuve du fait que le NoSQL, plus que le retour d'une ancienne mode, est une vraie alternative à des SGBD classiques, en voici une : certains SGBD, comme PostgreSQL pour ne pas le citer, ayant pris conscience du bénéfice d'une approche NoSQL, intègrent de tels composants en leur sein. ■

Références

[1] Redis : http://redis.io/

[2] Memcached : http://memcached.org/

[3] Cassandra : https://cassandra.apache.org/

[4] CouchDB : https://couchdb.apache.org/

[5] MongoDB : http://www.mongodb.org/

[6] Neo4J : http://www.neo4j.org/

[7] Infogrid : http://infogrid.org/trac/

[8] Les drivers MongoDB :http://docs.mongodb.org/ecosystem/drivers/

[9] GitHub de Redis : https://github.com/antirez/redis/tree/unstable

GNU LINUX PRATIQUE

DÉCOUVRIR, COMPRENDRE ET UTILISER LINUX

N° 78 JUILLET/AOÛT

À NE PAS RATER !

FAITES PARLER VOTRE SYSTÈME !

DISPONIBLE DÈS LE 28 JUIN
CHEZ VOTRE MARCHAND DE JOURNAUX
ET SUR : www.ed-diamond.com

LES MODULES PYTHON POUR LIRE ET ÉCRIRE DANS UNE BASE DE DONNÉES

par Tristan Colombo

Le langage Python, qui dispose de milliers de modules permettant d'étendre ses possibilités, propose bien sûr des modules pour accéder à différents systèmes de gestion de bases de données. Dans cet article, nous allons étudier des modules permettant d'accéder à un SGBD PostgreSQL, MySQL/MariaDB, ou encore SQLite. Vous verrez que, comme d'habitude avec Python, cela ne représente pas une grande difficulté. Pour finir, nous découvrirons un aperçu de ce qu'est un ORM (Object-Relational Mapping) avec le module SQLAlchemy.

Les modules utilisés dans cet article devront bien sûr être installés. Quel que soit le module, vous pourrez l'installer en suivant l'une des deux méthodes suivantes :

1. À partir des dépôts de votre distribution : pour les distributions basées sur Debian, le nom du paquetage commencera par **`python-`** et sera suivi par le nom du module.

2. En utilisant la commande **`pip install`** suivie du nom du module : cette technique est celle qui est utilisée avec les environnements virtuels et permet donc de bien cibler les modules (et leurs versions) nécessaires pour un projet.

Python 2.7 et Python 3.3

Actuellement, près de 80% des modules ont migré vers Python 3 : c'est beaucoup, mais c'est bien peu... Dans un projet un peu important, vous pouvez à tout moment avoir besoin d'un module écrit en utilisant la branche 2.7 et il faudra alors basculer entièrement le projet en 2.7. La solution la plus convenable à l'heure actuelle est de continuer à développer en Python 2.7, tout en utilisant un maximum de syntaxes compatibles 2.7/3.3 et en ajoutant des commentaires pour aider à la migration future du projet.

Les modules majeurs tels que **`numpy`**, **`scipy`** et **`matplotlib`** sont disponibles en Python 3, donc on peut espérer un passage définitif à cette branche d'ici quelques mois. En attendant, tous les exemples de cet article sont donnés en Python 2.7.

1 Fonctionnement général

Les modules Python d'accès aux bases de données sont bien conçus : ils fonctionnent tous en suivant le même schéma et pratiquement la même syntaxe. Voici les différentes étapes à suivre, accompagnées d'un pseudo-code générique :

1. Création d'une connexion avec une base de données :

```
db = connexion(serveur, utilisateur, mot_de_passe, nom_base_de_données)
```

2. Création d'un curseur dans la base pour pouvoir manipuler les données :

```
cursor = db.creer_curseur()
```

3. Exécution d'une requête :

```
resultat = cursor.execute("ma_requete_SQL")
```

4. Traitement de la requête et éventuellement validation des modifications :

- Exemple pour un affichage des données extraites :

```
pour chaque ligne de resultat :
  afficher ligne
```

- Exemple pour une validation d'une modification :

```
db.commit()
```

5. Fermeture du curseur, puis de la base :

```
cursor.fermeture()
db.fermeture()
```

2 psycopg2 pour PostgreSQL

Dans les exemples de cet article, nous allons encore travailler avec les tables **Client**, **Article** et **Commande**. Comme les exemples de l'article portant sur le langage SQL étaient entièrement écrits sous MySQL, je vais un peu plus détailler la création des bases sous PostgreSQL. Nous allons ici créer la table **Article** et y ajouter trois enregistrements. Tout d'abord, nous nous connectons en utilisant un nom d'utilisateur, puis nous créons la base et nous la déclarons comme base par défaut :

```
tristan=# CREATE DATABASE siteMarchand OWNER tristan;
CREATE DATABASE
tristan=# \c sitemarchand
You are now connected to database "sitemarchand" as user "tristan".
sitemarchand=# CREATE TABLE Article (
idArticle SERIAL NOT NULL,
description VARCHAR(100) NOT NULL,
puHT REAL NOT NULL,
PRIMARY KEY (idArticle));
NOTICE:  CREATE TABLE will create implicit sequence "article_idarticle_seq" for serial column
"article.idarticle"
NOTICE:  CREATE TABLE / PRIMARY KEY will create implicit index "article_pkey" for table
"article"
CREATE TABLE
```

Au niveau des différences entre MySQL et PostgreSQL, on peut voir que le **USE** n'existe pas et qu'il est remplacé par **\c**. Notez que PostgreSQL convertit les identifiants en caractères minuscules et donc, que bien qu'ayant demandé la création d'une base de données **siteMarchand**, la base effectivement créée se nommera **sitemarchand**. Autre différence : un champ en auto-incrémentation (**AUTO_INCREMENT**) est ici un type particulier nommé **SERIAL**. L'insertion des enregistrements se fait ensuite de la même manière qu'avec MySQL :

```
sitemarchand=# INSERT INTO Article VALUES
sitemarchand-#     (1, 'Plex', 1024.99),
sitemarchand-#     (2, 'Batterie', 5557.86),
sitemarchand-#     (3, 'Mine sonique', 12000);
INSERT 0 3
```

Utilisons maintenant **psycopg2** pour accéder à ces informations depuis Python :

```
01: # -*- encoding:utf-8 -*-
02:
03: import psycopg2
04: import sys
05:
06: db = None
07:
08: try:
09:     db = psycopg2.connect(
10:         host = "localhost",
11:         user = "tristan",
12:         password = "motDePasse",
13:         dbname = "sitemarchand")
14:     cursor = db.cursor()
15:
16:     query = """SELECT * FROM Article"""
17:     cursor.execute(query)
18:
19:     print 'Résultats :'
20:     for line in cursor.fetchall():
21:         print '  - %s' % line[1]
22:
23: except psycopg2.DatabaseError, e:
24:     print 'Error %s' % e
25:     sys.exit(1)
26:
27: finally:
28:     if db:
29:         cursor.close()
30:         db.close()
```

Nous retrouvons dans ce code les 5 étapes du fonctionnement général décrites précédemment :

1. La connexion dans les lignes 9 à 13 ;
2. La création du curseur en ligne 14 ;
3. L'exécution de la requête dans les lignes 16 et 17. La méthode **execute()** de **psycopg2** admet en second paramètre un tuple contenant des données qui peuvent être insérées dans la chaîne de caractères passée en premier paramètre comme lors d'un formatage. Ce mécanisme permet de protéger contre les injections SQL si des données constituant la requête sont fournies par un utilisateur.

4. Le traitement des données dans les lignes 19 à 21. La méthode **fetchall()** appliquée au curseur permet de récupérer une liste de tuples correspondant aux lignes extraites.
5. La fermeture du curseur et de la base dans les lignes 29 et 30.

Un traitement des erreurs a été ajouté sous la forme d'un bloc **try**/**except**/**finally** avec affichage du message d'erreur renvoyé par **pyscopg2** en ligne 24.

L'appel de ce script nous affiche la liste des articles présents dans la table **Article** de la base **siteMarchand** :

```
Résultats :
  - Plex
  - Batterie
  - Mine sonique
```

Sur le canevas de ce code Python, il suffit de modifier la requête SQL pour pouvoir effectuer toutes les actions souhaitées sur la base. Les méthodes **commit()** et **rollback()** appliquées sur un curseur permettent respectivement de valider et d'annuler des transactions.

3 mysqldb pour MySQL / MariaDB

Ici encore, nous retrouvons nos 5 étapes avec un code à peine différent de celui utilisé pour **psycopg2** :

```
01: # -*- encoding:utf-8 -*-
02:
03: import MySQLdb
04: import sys
05:
06: db = None
07:
08: try:
09:     db = MySQLdb.connect(
10:         "localhost",
11:         "tristan",
12:         "motDePasse",
13:         "siteMarchand")
14:     cursor = db.cursor()
15:
16:     query = """SELECT * FROM Article"""
17:     result = cursor.execute(query)
18:
19:     print 'Il y a %d résultats :' % (result,)
20:     for line in cursor.fetchall():
21:         print '  - %s' % line[1]
22:
23: except MySQLdb.DatabaseError, e:
24:     print 'Error %s' % e
25:     sys.exit(1)
26:
27: finally:
28:     if db:
29:         cursor.close()
30:         db.close()
```

Les lignes en rouge sont les lignes modifiées par rapport au code précédent (bien sûr je ne compte pas les changements de nom de module). Dans les lignes 9 à 13, nous n'avons pas nommé les paramètres et en ligne 17, la méthode **execute()** qui ne renvoyait rien nous donne ici le nombre de lignes extraites (ce nombre est ensuite utilisé en ligne 18 pour être affiché). Le résultat est bien le même que précédemment :

```
$ python MySQLdb_test.py
Il y a 3 résultats :
  - Plex
  - Batterie
  - Mine sonique
```

Si des données sont fournies par des utilisateurs pour créer une requête, **MySQLdb** fournit la méthode **escape_string()** qui protégera votre base d'une injection SQL.

4 sqlite3 pour SQLite

Jusqu'ici, nous n'avions pas parlé de SQLite. Il s'agit d'un SGBD un peu particulier, puisqu'il ne se présente pas sous une architecture client-serveur, mais qu'il est directement intégré aux applications. Les données sont stockées dans un seul fichier.

En Python, le module **sqlite3** permettant d'interagir avec une base SQLite est intégré aux modules de la bibliothèque standard. Vous n'aurez donc aucune installation particulière à exécuter.

Comme la base de données n'existe pas, il va falloir la créer. Pour pouvoir comparer avec les codes précédents, nous allons donc réaliser deux étapes distinctes : une étape de création de la table **Article**, puis une seconde étape d'extraction des données, que nous comparerons avec les deux codes utilisant **psycopg2** et **MySQLdb**. Commençons par créer notre table :

```
01: # -*- encoding:utf-8 -*-
02:
03: import sqlite3
04: import sys
05: import os
06:
07: db = None
08:
09: if not os.path.exists('maBase.db'):
10:     with open('maBase.db', 'w'):
11:         pass
12: else:
13:     print "File maBase.db already exists"
14:
15: try:
16:     db = sqlite3.connect('maBase.db')
17:     cursor = db.cursor()
18:
19:     cursor.execute("""CREATE TABLE Article (
20:         idArticle INTEGER NOT NULL,
```

```
21:         description VARCHAR(100) NOT NULL,
22:         puHT REAL NOT NULL,
23:         PRIMARY KEY (idArticle));""")
24:
25:     cursor.execute("INSERT INTO Article VALUES (1, 'Plex', 1024.99)")
26:     cursor.execute("INSERT INTO Article VALUES (2, 'Batterie', 5557.86)")
27:     cursor.execute("INSERT INTO Article VALUES (3, 'Mine sonique', 12000);")
28:     db.commit()
29:
30: except sqlite3.DatabaseError, e:
31:     print 'Error %s' % e
32:     sys.exit(1)
33:
34: finally:
35:     if db:
36:         cursor.close()
37:         db.close()
```

Dans les lignes 9 à 13, nous testons l'existence du fichier de base de données **maBase.db** : si celui-ci n'existe pas, nous le créons (lignes 10 et 11), sinon nous affichons un message (ligne 13). Nous nous connectons ensuite de manière classique à la base et nous créons un curseur (lignes 16 et 17). Dans les lignes 19 à 23, nous exécutons la requête permettant de créer la table. Notez l'absence d'un mot-clé indiquant l'auto-incrément dans la définition du champ **idArticle** en ligne 20 : avec SQLite, une clé primaire de type entier est en mode auto-incrément par défaut.

Pour ajouter des enregistrements dans la table, nous exécutons plusieurs requêtes **INSERT** de manière consécutive dans les lignes 25 à 27. Il ne faut surtout pas oublier la ligne 28 effectuant la validation des modifications, SQLite ne fonctionnant pas en mode de validation automatique. La suite du code permet de gérer les erreurs (lignes 30 à 32) et de fermer proprement le curseur et la base (lignes 34 à 37).

Cette partie est déjà très semblable à ce que nous avons vu jusque-là avec les deux autres SGBD. L'extraction de données le sera encore plus :

```
01: # -*- encoding:utf-8 -*-
02:
03: import sqlite3
04: import sys
05:
06: db = None
07:
08: try:
09:     db = sqlite3.connect('maBase.db')
10:     cursor = db.cursor()
11:
12:     query = """SELECT * FROM Article;"""
13:     cursor.execute(query)
14:     db.commit()
15:
16:     print 'Résultats:'
17:     for line in cursor.fetchall():
18:         print '  - %s' % line[1]
19:
20: except sqlite3.DatabaseError, e:
21:     print 'Error %s' % e
22:     sys.exit(1)
23:
24: finally:
25:     if db:
26:         cursor.close()
27:         db.close()
```

Si nous comparons ce code avec celui permettant d'extraire des données dans une base PostgreSQL à l'aide de **psycopg2**, il n'y a qu'une ligne – indiquée en rouge – qui diffère : la connexion à la base.

Sur ces trois modules, nous avons pu voir que la syntaxe Python pour interagir avec des SGBD PostgreSQL, MySQL/MariaDB, ou SQLite était la même avec pour seule différence flagrante la création de la connexion. Il existe bien sûr une autre différence, mais qui n'est pas le fait des modules Python : la syntaxe SQL varie entre les SGBD et un changement de SGBD en cours de développement ne sera pas toujours simple...

5 Aperçu d'un ORM avec SQLAlchemy

Je vous propose ici un exemple très simple d'utilisation d'un ORM avec le module SQLAlchemy ; mais, avant de commencer à mettre en place un ORM, encore faudrait-il savoir de quoi il s'agit.

Un ORM va établir un lien entre des objets côté programme et des tables côté SGBD. Le programme n'exécutera plus directement des requêtes SQL, mais travaillera sur les objets. L'intérêt de ce genre d'approche est d'obtenir un niveau d'abstraction plus élevé et de rendre le code entièrement indépendant du SGBD choisi. En effet, comme le code SQL est généré par l'ORM, celui-ci s'adapte au SGBD et si vous décidez en cours de développement de changer de SGBD, vous n'aurez pas à modifier votre code : tout fonctionnera correctement.

Un autre avantage est de ne pas avoir à écrire - donc éventuellement à connaître - le langage SQL. Cet avantage est discutable, car il est toujours préférable de comprendre ce que fait le code : en cas de ralentissement, cela permet de savoir où commencer les recherches. Sachez enfin que même

si les ORM intègrent une méthode permettant d'exécuter du code SQL, il est toujours préférable d'utiliser les fonctions définies dans l'ORM pour ne pas perdre l'indépendance du code vis-à-vis des SGBD.

D'un point de vue pratique, l'installation de SQLAlchemy pourra se faire soit depuis les dépôts de votre distribution de la même manière que pour les modules précédents, soit en utilisant la commande **pip** :

```
sudo pip install sqlalchemy
```

Pour commencer, nous allons créer la table **Article**. Cette création ne passera que par du code Python en définissant une classe **Article** (fichier **Article.py**) :

```
01: # -*- encoding:utf-8 -*-
02:
03: from sqlalchemy import create_engine
04: from sqlalchemy.ext.declarative import
declarative_base
05: from sqlalchemy import Column, Integer,
String, Float
06:
07: db = create_engine('sqlite:///maBase.
db', echo = True)
08: Base = declarative_base()
09:
10: class Article(Base):
11:     __tablename__ = 'Article'
12:
13:     idArticle = Column(Integer,
primary_key = True)
14:     description = Column(String)
15:     puHT = Column(Float)
16:
17:     def __init__(self, description,
puHT):
18:         self.description = description
19:         self.puHT = puHT
20:
21: Base.metadata.create_all(db)
```

Dans les lignes 3 à 5, nous importons les éléments du module **sqlalchemy** dont nous aurons besoin. En ligne 7, nous créons la connexion à la base. Ici, il s'agit d'une base SQLite et le fichier de base de données sera **maBase.db**. La ligne 8 permet de créer l'objet qui servira de lien avec la base de données. Dans les lignes 10 à 19, nous définissons la classe **Article** qui est le reflet de la table **Article** attendue. Les colonnes de la table sont décrites à l'aide d'attributs de classe instanciés en tant qu'objets de type **Column**. Le constructeur de cette classe admet, entre autres, comme paramètres le type de la colonne (ici **Integer**, **String**, ou **Float**) et **primary_key** qui indique à l'aide d'une valeur booléenne si la colonne doit être utilisée en tant que clé primaire. Le constructeur de la classe **Article** (lignes 17 à 19) nous permettra d'ajouter des enregistrements en indiquant une description et un prix unitaire hors taxes.

Au lancement du script, comme nous avons activé l'affichage des messages, nous obtiendrons les informations suivantes :

```
2013-05-25 13:32:43,341 INFO sqlalchemy.engine.base.Engine PRAGMA table_info("Article")
2013-05-25 13:32:43,341 INFO sqlalchemy.engine.base.Engine ()
2013-05-25 13:32:43,342 INFO sqlalchemy.engine.base.Engine
CREATE TABLE "Article" (
        "idArticle" INTEGER NOT NULL,
        description VARCHAR,
        "puHT" FLOAT,
        PRIMARY KEY ("idArticle")
)

2013-05-25 13:32:43,342 INFO sqlalchemy.engine.base.Engine ()
2013-05-25 13:32:43,469 INFO sqlalchemy.engine.base.Engine COMMIT
```

Nous pouvons donc voir la traduction sous forme de requête SQL de la création de la table **Article** que nous avions codée en Python. Si nous relançons ce script alors que la table existe déjà, celle-ci ne sera pas recréée :

```
2013-05-25 13:33:07,476 INFO sqlalchemy.engine.base.Engine PRAGMA table_info("Article")
2013-05-25 13:33:07,476 INFO sqlalchemy.engine.base.Engine ()
```

À ce stade, si vous souhaitez changer de SGBD, vous n'aurez que la ligne 7 à modifier. Supposons que nous souhaitions nous connecter à le base MySQL **siteMarchand**, le code sera alors :

```
07: db = create_engine('mysql://utilisateur:motDePasse@localhost/siteMarchand', echo = True)
```

Il est donc très simple de migrer d'un SGBD à un autre, à partir du moment où l'on utilise un ORM.

Voici maintenant comment utiliser la table que nous venons de créer sans écrire une ligne de SQL. Nous allons instancier des objets **Article** que nous « enregistrerons » dans la base et nous extrairons des données uniquement à l'aide de méthodes :

```
01: # -*- encoding:utf-8 -*-
02:
03: from sqlalchemy import create_engine
04: from sqlalchemy.orm import sessionmaker
05: from Article import *
06:
07: db = create_engine('sqlite:///maBase.db', echo = True)
08:
09: Session = sessionmaker(bind = db)
10: session = Session()
11:
12: new_article = Article('Plex', 1024.99)
13: session.add(new_article)
14: session.commit()
15:
```

```
16: new_articles = [
17:     Article("Batterie", 5557.86),
18:     Article("Mine sonique", 12000),
19: ]
20: session.add_all(new_articles)
21: session.commit()
22:
23: result = session.query(Article).
filter(Article.puHT < 10000)
24:
25: print "Résultat :"
26: for line in result:
27:     print "  - %s" % (line.description)
```

Nous créons la connexion en ligne 7 de la même manière que précédemment. Les lignes 9 et 10 permettent de créer une « session » qui sera liée à notre connexion. En fait, la session de SQLAlchemy est le curseur que nous avons déjà utilisé dans les autres modules. Pour l'ajout d'enregistrements dans la table **Article**, deux méthodes ont été utilisées :

- Dans les lignes 12 à 14, nous n'ajoutons qu'une seule ligne en créant un objet **Article** et en précisant les valeurs des champs en tant que paramètres ;
- Dans les lignes 16 à 21, nous créons une liste d'objets **Article**.

Ces deux méthodes contiennent en dernières lignes un ajout dans la base utilisant respectivement **session.add()** et **session.add_all()** et la validation des modifications par **session.commit()**.

Pour extraire les données de la base (ligne 23), nous créons la requête pas à pas à l'aide de méthodes : **session.query(Article)** indique une requête dans la table **Article** et **filter(Article.puHT < 10000)** indique une contrainte sur le prix unitaire. En parcourant la variable **resultat**, nous obtenons des objets représentant les lignes et dont les attributs portent le nom des colonnes (lignes 25 à 27). Ce système est beaucoup plus lisible que l'extraction des données d'une liste en précisant un indice...

Lors de l'exécution du code, nous pouvons voir les requêtes générées et exécutées :

```
2013-05-25 14:29:04,151 INFO sqlalchemy.engine.base.Engine PRAGMA table_info("Article")
2013-05-25 14:29:04,151 INFO sqlalchemy.engine.base.Engine ()
2013-05-25 14:29:04,152 INFO sqlalchemy.engine.base.Engine BEGIN (implicit)
2013-05-25 14:29:04,153 INFO sqlalchemy.engine.base.Engine INSERT INTO "Article"
(description, "puHT") VALUES (?, ?)
2013-05-25 14:29:04,153 INFO sqlalchemy.engine.base.Engine ('Plex', 1024.99)
2013-05-25 14:29:04,153 INFO sqlalchemy.engine.base.Engine COMMIT
2013-05-25 14:29:04,299 INFO sqlalchemy.engine.base.Engine BEGIN (implicit)
2013-05-25 14:29:04,299 INFO sqlalchemy.engine.base.Engine INSERT INTO "Article"
(description, "puHT") VALUES (?, ?)
2013-05-25 14:29:04,299 INFO sqlalchemy.engine.base.Engine ('Batterie', 5557.86)
2013-05-25 14:29:04,300 INFO sqlalchemy.engine.base.Engine INSERT INTO "Article"
(description, "puHT") VALUES (?, ?)
2013-05-25 14:29:04,300 INFO sqlalchemy.engine.base.Engine ('Mine sonique', 12000.0)
2013-05-25 14:29:04,300 INFO sqlalchemy.engine.base.Engine COMMIT
Résultat :
2013-05-25 14:29:04,397 INFO sqlalchemy.engine.base.Engine BEGIN (implicit)
2013-05-25 14:29:04,397 INFO sqlalchemy.engine.base.Engine SELECT "Article"."idArticle" AS
"Article_idArticle", "Article".description AS "Article_description", "Article"."puHT" AS
"Article_puHT"
FROM "Article"
WHERE "Article"."puHT" < ?
2013-05-25 14:29:04,398 INFO sqlalchemy.engine.base.Engine (10000,)
  - Plex
  - Batterie
```

Conclusion

Quelques lignes suffisent en Python pour se connecter et extraire ou ajouter des données dans une base. Il faut simplement choisir le SGBD avec lequel on souhaite travailler, sélectionner le bon module et écrire les requêtes SQL. Si vous vous orientez vers une utilisation intensive des bases de données, l'apprentissage d'un ORM rendra votre code portable et plus aisément maintenable. Ce choix vous demandera forcément plus de travail et de temps au départ, mais pourra s'avérer rentable très rapidement. ■

TRAVAILLER EN C AVEC UNE BASE DE DONNÉES

par Tristan Colombo

En C aussi on peut travailler avec des bases de données, et contrairement à ce que l'on pourrait penser, ce n'est pas si compliqué : chaque système de gestion de base de données propose une API qui facilite la communication entre le programme et la base.

Dans cet article, nous reprendrons notre sempiternel exemple consistant à afficher la liste des articles de la table **Article** stockée dans la base **siteMarchand**. Nous testerons trois API pour communiquer avec trois SGBD différents : PostgreSQL, MySQL, puis SQLite.

1 PostgreSQL

Pour accéder à une base de données PostgreSQL depuis un programme C, nous allons utiliser la bibliothèque **libpq-fe**. Cette bibliothèque peut être installée depuis un gestionnaire de paquetages. Pour les distributions basées sur Debian, il s'agit du paquet **libpq-dev** :

```
~$ sudo aptitude install libpq-dev
```

Ici, contrairement aux autres méthodes vues en Python et Java, le curseur n'est pas déclaré explicitement. Les autres étapes restent semblables :

```
01: #include <libpq-fe.h>
02: #include <stdlib.h>
03:
04: int main(void)
05: {
06:     const char *conninfo;
07:     PGconn *connection;
08:     PGresult *result;
09:     int i, cpt;
10:
11:     // Connexion
12:     conninfo = "hostaddr = '127.0.0.1' \
13:                 port = '' \
14:                 dbname = 'sitemarchand' \
15:                 user='tristan' \
16:                 password='motDePasse'";
17:     connection = PQconnectdb(conninfo);
18:     if (!connection)
19:     {
20:         fprintf(stderr, "La connexion est impossible.\n\n");
21:         exit(1);
22:     }
23:     if (PQstatus(connection) != CONNECTION_OK)
24:     {
25:         fprintf(stderr, "Erreur dans les paramètres de connexion.\n\n");
26:         exit(2);
27:     }
28:
29:     // Requête
30:     result = PQexec(connection, "SELECT * FROM Article");
31:     if (PQresultStatus(result) != PGRES_TUPLES_OK)
32:     {
33:       fprintf(stderr, "La requête a échoué : %s", PQerrorMessage(connection));
34:       PQclear(result);
35:       PQfinish(connection);
36:       exit(3);
37:     }
38:
39:     cpt = PQntuples(result);
40:     printf("Il y a %d résultats :\n", cpt);
41:     for (i = 0; i < cpt; i++)
42:     {
43:         printf("  - %s\n", PQgetvalue(result, i, 1));
44:     }
45:     PQclear(result);
46:
47:     PQfinish(connection);
48:
49:     return 0;
50: }
```

En ligne 1, nous importons la bibliothèque **libpq-fe** permettant de dialoguer avec un SGBD PostgreSQL. En ligne 2, nous importons **stdlib** pour pouvoir utiliser la fonction **exit()**. Nous définissons ensuite une seule fonction qui sera le programme principal (lignes 4 à 50). Dans les lignes 6 à 9, nous déclarons les variables dont nous aurons besoin dans notre code :

- une chaîne **conninfo** (ligne 6) pour stocker les paramètres de connexion à la base ;
- un pointeur **connection** de type **PGconn** (ligne 7) pour établir la connexion ;
- un pointeur **result** de type **PGresult** (ligne 8) pour stocker le résultat des requêtes ;
- deux entiers **i** et **cpt** (ligne 9), qui permettront respectivement de parcourir une boucle et de stocker le nombre de lignes extraites de la base.

La création de la connexion a lieu dans les lignes 11 à 27. Nous commençons par définir les paramètres de connexion au sein d'une chaîne de caractères (lignes 12 à 16) dans laquelle nous donnons l'adresse du serveur PostgreSQL, le port (ici nous utilisons le port par défaut), le nom de la base de données que nous allons utiliser, le nom et le mot de passe de l'utilisateur que nous allons utiliser pour nous connecter. Ces informations sont ensuite utilisées en ligne 17 pour établir la connexion. En cas d'erreur, si la connexion est impossible pour différentes raisons, nous affichons un message d'erreur et quittons le programme (lignes 18 à 27).

En ligne 30, une fois la connexion réalisée, nous effectuons une simple requête permettant d'extraire toutes les lignes de la table **Article**. Si la requête échoue, nous affichons un message d'erreur, refermons proprement la connexion à la base et quittons le programme (lignes 31 à 37). Si la requête a été correctement exécutée, nous récupérons le nombre de lignes extraites grâce à la fonction **PQntuples()** en ligne 39, puis nous affichons les résultats dans les lignes 40 à 44.

Notez que l'on accède aux champs de chaque ligne extraite de la table par la fonction **PQgetvalue()**, qui prend pour paramètres la variable contenant les résultats, le numéro de ligne et le numéro de colonne (tous les deux commençant de façon normale à 0).

En ligne 45, nous libérons la mémoire occupée par le résultat de notre requête, puis nous refermons la connexion avec la base (ligne 47) et nous terminons le programme (ligne 49).

Pour compiler ce code, nous utiliserons un fichier **Makefile** (voir encadré pour quelques rappels sur l'écriture de tels fichiers) :

Le Makefile

L'utilitaire **make** permet de re-compiler automatiquement les fichiers modifiés et leurs dépendances. Pour déclarer les règles de compilation, on crée un fichier **Makefile** utilisant une syntaxe particulière. La structure générale d'un tel fichier est :

```
cible : dépendances
  commandes
```

Il est possible de déclarer des variables qui seront ensuite utilisées en les encadrant de parenthèses et en les préfixant par un **$** :

```
CC = g++
CFLAGS = -Wall

test.o : test.c
  $(CC) $(CFLAGS) -c test.c -o test.o
```

La ligne précédente est traduite en :

```
g++ -Wall -c test.c -o test.o
```

Il existe des variables spéciales permettant de raccourcir l'écriture des règles :

- **$<** : nom de la première dépendance ;
- **$^** : liste de toutes les dépendances ;
- **$@** : nom de la cible.

La règle précédente peut ainsi s'écrire :

```
test.o : test.c
  $(CC) $(CFLAGS) -c $< -o $@
```

Enfin, le caractère **%** est un caractère joker permettant d'écrire des règles généralistes :

```
%.o : %.c %.h
  $(CC) $(CFLAGS) -c $< -o $@
```

Cette règle fonctionnera pour **test.o : test.c test.h**, mais également pour **pgc.o : pgc.c pgc.h**, etc.

```
01: CC       = gcc
02: LINKLIBS = -I/usr/include/postgresql
03: CFLAGS   = -g
04: LDLIBS   = -L/usr/include/postgresql/ -lpq
05: OPTS     = -Wall -O5
06:
07: pg_test: pg_test.o
08:
09: pg_test.o: pg_test.c
10:  $(CC) $(OPTS) $(CFLAGS) $(LINKLIBS) $(LDLIBS) -c $< -o $@
11:
12: clean:
13:  rm -rf *.o
```

Il n'y a plus qu'à compiler en tapant la commande **make**, puis à exécuter le code en lançant **pg_test**.

Pour finir, si vous souhaitez utiliser une requête préparée, il faudra vous tourner vers la fonction **PqexecPrepared()** en lieu et place de **PQexec()**.

2 MySQL

L'utilisation de l'API MySQL est très semblable à celle de PostgreSQL. Il faudra, là encore, installer la bibliothèque contenue dans un paquetage :

```
~$ sudo aptitude install libmysqlclient-dev
```

J'ai volontairement respecté la même structure que dans l'exemple précédent pour que les similarités des deux codes soient évidentes :

```
01: #include <mysql.h>
02: #include <stdlib.h>
03:
04: int main(void)
05: {
06:     const char *server, *user, *password, *dbname;
07:     MYSQL connection;
08:     MYSQL_RES *result = NULL;
09:     MYSQL_ROW row;
10:     int i, cpt;
11:
12:     // Connexion
13:     server   = "127.0.0.1";
14:     user     = "tristan";
15:     password = "motDePasse";
16:     dbname   = "siteMarchand";
17:     mysql_init(&connection);
18:     mysql_options(&connection, MYSQL_READ_DEFAULT_GROUP, "option");
19:     if (!mysql_real_connect(&connection, server, user, password,
20:                             dbname,0,NULL,0))
21:     {
22:         fprintf(stderr, "La connexion est impossible.\n\n");
23:         exit(1);
24:     }
25:
26:     //Requête
27:     mysql_query(&connection, "SELECT * FROM Article");
28:     result = mysql_use_result(&connection);
29:
30:     //On récupère le nombre de champs
31:     cpt = mysql_num_fields(result);
32:     printf("Il y a %d résultats :\n", cpt);
33:     for (i = 0; i < cpt; i++)
34:     {
35:       row = mysql_fetch_row(result);
36:       printf("  - %s\n", row[1]);
37:     }
38:     mysql_free_result(result);
39:
40:     mysql_close(&connection);
41:
42:     return 0;
43: }
```

Dans les lignes 1 et 2, nous importons les bibliothèques nécessaires : API MySQL et **stdlib** (toujours pour la fonction **exit()**). Dans les lignes 6 à 10, nous déclarons les variables qui nous seront nécessaires. Les paramètres de connexion sont stockés dans plusieurs chaînes de caractères (contrairement à la version PostgreSQL), d'où les quatre variables de la ligne 6.

En ligne 7, la variable de connexion est cette fois de type **MYSQL**. En ligne 8, nous voyons que les lignes seront extraites dans une variable de type **MYSQL_RES** et, nouveauté en ligne 9, une variable de type **MYSQL_ROW** nous permettra de parcourir les résultats.

Dans les lignes 13 à 16, nous définissons les paramètres de connexion, puis la connexion proprement dite est réalisée en trois étapes dans les lignes 17 à 19 : initialisation, définition des options et connexion à la base à l'aide de la fonction **mysql_real_connect()**. En cas d'erreur, nous affichons un message et quittons le programme (lignes 22 et 23).

Les informations sont extraites de la base dans les lignes 27 et 28, puis affichées dans les lignes 31 à 37. Ici, les résultats sont récupérés ligne à ligne grâce à la fonction **mysql_fetch_row()** et stockés dans la variable **row**. Pour obtenir la valeur d'une colonne, il faut ensuite indiquer l'index souhaité en utilisant **row** (voir ligne 36).

Enfin, les lignes 38 et 40 libèrent la mémoire occupée par le résultat de la requête et referment la connexion avec la base.

En ce qui concerne la compilation, le fichier **Makefile** est lui aussi très semblable au précédent ; il faut simplement modifier les chemins vers la bibliothèque **mysql** (notés en rouge) :

```
01: CC       = gcc
02: LINKLIBS = -I/usr/include/mysql
03: CFLAGS   = -g
04: LDLIBS   = -L/usr/include/mysql/ -lmysqlclient
05: OPTS     = -Wall -O5
06:
07: mysql_test: mysql_test.o
08:
09: mysql_test.o: mysql_test.c
10:  $(CC) $(OPTS) $(CFLAGS) $(LINKLIBS) $(LDLIBS) -c $< -o $@
11:
12: clean:
13:  rm -rf *.o
```

Notez que cette technique fonctionnera également pour les bases MariaDB.

3 SQLite

Pour finir, voici un petit exemple d'utilisation de l'API SQLite3. Comme précédemment, le paquetage de développement est indispensable :

```
~$ sudo aptitude install libsqlite3-dev
```

Ici, la structure du code va être légèrement différente de ce que nous avions pu voir avec PostgreSQL et MySQL :

```
01: #include <sqlite3.h>
02: #include <stdlib.h>
03: #include <stdio.h>
04: #include <string.h>
05:
06: int main(void)
07: {
08:     sqlite3 *connection;
09:     sqlite3_stmt *result;
10:     const char *sql;
11:
12:     // Connexion
13:     if (sqlite3_open("maBase.db", &connection))
14:     {
15:         fprintf(stderr, "Impossible d'ouvrir la base : %s", sqlite3_errmsg(connection));
16:         exit(1);
17:     }
18:
19:     // Requête
20:     sql = "SELECT * FROM Article";
21:     if (sqlite3_prepare(connection, sql, strlen(sql) + 1, &result, NULL) != SQLITE_OK)
22:     {
23:         fprintf(stderr, "Erreur dans la requête");
24:         exit(2);
25:     }
26:     while (sqlite3_step(result) == SQLITE_ROW)
27:     {
28:         printf("  - %s\n", sqlite3_column_text(result, 1));
29:     }
30:
31:     sqlite3_finalize(result);
32:
33:     sqlite3_close(connection);
34:
35:     return 0;
36: }
```

Dans les lignes 1 à 4, nous chargeons les différentes bibliothèques dont nous aurons besoin y compris, bien sûr, **sqlite3**. Les variables sont déclarées dans les lignes 8 à 10 : un pointeur de type **sqlite3** pour la connexion à la base, un pointeur de type **sqlite3_stmt** pour stocker les données extraites et une chaîne de caractères pour stocker notre requête.

La connexion à la base se fait dans les lignes 12 à 17, grâce à la fonction **sqlite3_open()** à laquelle nous fournissons le nom de la base (nom du fichier). En cas d'erreur, nous affichons un message et quittons le programme.

Dans les lignes 19 à 29, nous effectuons une requête dans la base. Cette requête est stockée dans la variable **sql** (ligne 20) puis, en ligne 21, nous l'exécutons et récupérons les résultats dans la variable **result** (notez que le troisième paramètre de la fonction **sqlite3_prepare()** correspond à la taille de la chaîne de caractères **sql** à laquelle on ajoute le caractère **\0**). En cas d'erreur, nous affichons un message et sortons du programme (lignes 23 et 24), sinon nous parcourons les résultats pour les afficher (lignes 26 à 29).

L'accès aux données de chaque ligne se fait à l'aide de la fonction **sqlite3_column_text()** à laquelle nous devons indiquer l'index de l'élément à récupérer (le premier élément étant toujours 0). Enfin, dans les lignes 31 et 33, nous libérons la mémoire occupée par le résultat de notre requête et nous fermons la connexion avec la base de données.

La compilation se fera simplement en liant la bibliothèque **sqlite3** :

```
~$ gcc -lsqlite3 sqlite_test.c
```

Conclusion

Quel que soit le SGBD choisi, l'interaction avec un programme en C reste logique (pour ne pas dire simple). Avec un minimum de maîtrise du langage (pointeurs et Makefile), on obtient des résultats avec relativement peu de lignes de code : plus qu'en Python, mais moins qu'en Java... Nous avons pu tester ces trois langages pour communiquer avec des SGBD, il ne vous reste plus qu'à faire votre choix ! ■

ACCÉDER À UNE BASE DE DONNÉES EN JAVA

par Tristan Colombo

Comme avec tout langage standard, les bases de données peuvent être utilisées depuis un code Java, plus ou moins simplement... En Java, on utilise une API standard pour accéder à tous les SGBD. Il s'agit de JDBC pour Java DataBase Connectivity. L'avantage de cette solution est l'intégration native dans Java SE.

Il suffit donc de disposer des bons drivers pour se connecter au SGBD choisi et de l'indiquer lors de la connexion à la base. Les étapes pour accéder à une base de données seront sensiblement les mêmes qu'en Python :

1. Chargement du driver JDBC correspondant au SGBD sélectionné,
2. Ouverture de la connexion,
3. Création du curseur,
4. Exécution d'une requête,
5. Traitement des données,
6. Fermeture du curseur et de la base.

Il y a donc une étape supplémentaire : le chargement du driver. Cette étape est essentielle : sans elle, rien ne pourra fonctionner. Dans cet article, je détaillerai donc l'installation des drivers, avant de voir comment utiliser Java et JDBC puis, nous finirons encore par l'aperçu d'un ORM : Hibernate.

1 L'installation des drivers

En fonction de votre SGBD, vous devrez télécharger le driver (ou connecteur) approprié. Il suffit de vous rendre sur le site officiel de votre SGBD et de télécharger un fichier :

- pour MySQL : http://dev.mysql.com/downloads/connector/j/. Le fichier **mysql-connector-java-5.1.25.tar.gz** contient le fichier **mysql-connector-java-5.1.25-bin.jar** ;
- pour PostgreSQL : http://jdbc.postgresql.org/download.html. Ici, le fichier n'est pas compressé. Il s'agit de **postgresql-9.2-1002.jdbc4.jar**.

Vous pouvez ensuite placer ces fichiers dans le répertoire de votre projet, ou dans le répertoire des bibliothèques Java (**/usr/share/java** normalement).

Si vous disposez d'une distribution basée sur Debian, vous pouvez installer ces drivers directement depuis le gestionnaire de paquetages :

```
~$ sudo aptitude install libjava-mysql libpg-java
```

Quelle que soit la méthode choisie, vous allez maintenant devoir modifier votre variable d'environnement **CLASSPATH** de manière à ce que Java puisse voir ces drivers. Vous pouvez effectuer la modification dans un terminal et celle-ci ne sera valable que pendant la durée d'ouverture de ce terminal, ou alors dans le fichier de configuration de votre shell et la modification sera persistante :

```
export CLASSPATH=$CLASSPATH:/usr/share/java/mysql.jar:/usr/share/java/postgresql
```

Bien sûr, si vous avez choisi d'utiliser un driver localement (dans le même répertoire que votre code Java), vous pouvez déclarer le **CLASSPATH** au moment de l'appel de votre programme :

```
~$ CLASSPATH=/home/login/monProjet/mysql-connector-java-5.1.25-bin.jar java monProgramme.java
```

Assurez-vous de bien avoir effectué cette étape, sans cela il vous sera impossible de vous connecter à un SGBD et vous obtiendrez le message d'erreur suivant :

```
SQLException: No suitable driver found for jdbc:mysql://localhost?siteMarchand
```

2 Utilisation de JDBC

Une fois les problèmes de drivers réglés, nous allons pouvoir nous attaquer au code. L'objectif sera toujours le même : se connecter à la base **siteMarchand** et afficher la liste des articles présents dans la table **Article**.

```
01: import java.sql.Connection;
02: import java.sql.DriverManager;
03: import java.sql.SQLException;
04: import java.sql.Statement;
05: import java.sql.ResultSet;
06:
07: public class JDBC_mysql
08: {
09:     public static void main(String[] args)
10:     {
11:         // Variables de connexion
12:         Connection conn = null;
13:         String host     = "localhost",
14:                dbname   = "siteMarchand",
15:                user     = "nomUtilisateur",
16:                password = "motDePasse";
17:         try
18:         {
19:             // Connexion à la base
20:             try
21:             {
22:                 Class.forName("com.mysql.jdbc.Driver").newInstance();
23:             }
24:             catch (Exception e)
25:             {
26:                 e.printStackTrace();
27:             }
28:             conn = DriverManager.getConnection(
29:                 "jdbc:mysql://" + host + "/" + dbname,
30:                 user,
31:                 password
32:             );
33:             System.out.println("Connected to database " + dbname + " on " +
34:                 host + "\n");
35:
36:             // Extraction des données
37:             Statement Stmt = conn.createStatement();
38:             ResultSet RS   = Stmt.executeQuery("SELECT * FROM Article");
39:
40:             // Traitement des données
41:             System.out.println("Résultats :");
42:             while (RS.next() != false)
43:             {
44:                 System.out.println("  - " + RS.getString(2));
45:             }
46:
47:             // Fermetures
48:             RS.close();
49:             Stmt.close();
50:             conn.close();
51:         }
52:         catch (SQLException ex)
53:         {
54:             System.out.println("SQLException: " + ex.getMessage());
55:             System.out.println("SQLState: " + ex.getSQLState());
56:             System.out.println("VendorError: " + ex.getErrorCode());
57:         }
58:     }
59: }
```

Les lignes 1 à 5 comportent les imports nécessaires au fonctionnement du code. Nous définissons ensuite une classe (ligne 7 à 59) qui ne contient qu'une seule méthode **main()** (lignes 9 à 58). Dans les lignes 11 à 16, nous définissons des variables qui vont nous permettre de nous connecter à la base de données. Les variables de type chaîne de caractères contiennent l'adresse du serveur de base de données, le nom de la base, ainsi que le nom de l'utilisateur et son mot de passe.

Le chargement du driver se fait en ligne 22. Si vous utilisez un SGBD autre que MySQL/MariaDB, c'est cette ligne qu'il faudra modifier en remplaçant « mysql » par le nom de votre SGBD.

La connexion effective se fait dans les lignes 28 à 32 en utilisant les variables contenant les paramètres de connexion. L'objet **conn** est l'objet de connexion à partir duquel nous allons créer un objet **Statement** (le curseur) en ligne 37. La requête SQL est exécutée en ligne 38 et les lignes extraites de la base sont placées dans un objet itérable **RS** de type **ResultSet**. Dans les lignes 42 à 45, nous parcourons cet objet pour afficher les noms des différents articles. Pour accéder au champ qui nous intéresse (la désignation), nous devons indiquer son index en tant que paramètre de la méthode **getString()**. Notez que les indices débutent ici à 1 et non à 0... Il faut ensuite refermer proprement tous les objets utilisés (lignes 48 à 50). Enfin, les lignes 52 à 57 correspondent au traitement des exceptions qui peuvent être levées en cours de traitement.

Il peut sembler très simple de transposer ce code pour d'autres SGBD et c'est vrai, puisqu'il suffit de modifier la connexion en ligne 22... Mais tout ce qui a été dit pour les modules Python reste vrai ici. Pour éviter des injections SQL, vous pourrez utiliser les requêtes préparées avec l'objet **PreparedStatement**. L'idée est de

déclarer une requête SQL dans laquelle vous laisserez en quelque sorte des espaces libres pour insérer des données. Les méthodes qui insèreront les données feront en sorte de protéger le code.

Autre intérêt des requêtes préparées : elles contiennent des instructions SQL compilées, ce qui améliore les performances en cas d'appels multiples. Voici un exemple arbitraire d'utilisation de requête préparée, qui va modifier un champ de la table **Article** :

```
String query = "UPDATE Article SET designation = ? WHERE
puHT = ?" ;
PreparedStatement prepared_query = conn.prepareStatement(query) ;
preparedStatement.setString(1, "Nouvelle description");
preparedStatement.setDouble(2, 12000.);
preparedStatement.executeUpdate();
```

La définition des valeurs insérées dans la requête se fait en précisant leur position (ou indice) dans une méthode **setString()**, **setDouble()**, etc.

3 Aperçu d'un ORM avec Hibernate

Après l'article sur Python et SQLAlchemy, Hibernate va vous sembler être une usine à gaz... Je vous rassure, ce n'est pas qu'une impression ! De toute façon, avec du Java, il fallait s'y attendre un peu... Donc, pour commencer, nous allons devoir installer quelques paquetages. Là, il vaut mieux disposer d'une distribution les proposant. Sur une distribution *Debian-like* il faudra installer :

```
~$ sudo aptitude install libhibernate3-java libhibernate-commons-
annotations-java libdom4j-java libslf4j-java libopenjpa-java
libjavassist.java
```

Une fois ces bibliothèques installées, il faut les rendre accessibles en les déclarant dans la variable d'environnement **CLASSPATH**, sans oublier le driver JDBC pour le SGBD que vous aurez choisi (ici MySQL/MariaDB) :

```
~$ export CLASSPATH=$CLASSPATH:/usr/share/java/mysql.jar:/usr/
share/java/hibernate3.jar:/usr/share/java/hibernate-commons-
annotations.jar:/usr/share/java/dom4j.jar:/usr/share/java/slf4j-
api.jar:/usr/share/java/slf4j-nop.jar:/usr/share/java/openjpa.
jar:/usr/share/java/hibernate:/usr/share/java/javassist.jar:/usr/
share/java/antlr.jar
```

Bien sûr, vous pouvez vous faciliter le travail en accédant arbitrairement à tout le répertoire **/usr/share/java** :

```
~$ export CLASSPATH=$CLASSPATH:/usr/share/java/*
```

La première source d'erreur vient d'être évitée... Passons à l'écriture de nos cinq fichiers. Pourquoi cinq ? Parce que quand on fait du Java, on ne rigole pas ! Nous avions l'air bien bête avec nos deux petits fichiers d'une vingtaine de lignes en Python ! Pour commencer, nous allons configurer Hibernate grâce au fichier XML **hibernate.cfg.xml** :

```
01: <!DOCTYPE hibernate-configuration PUBLIC
02:  "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
03:  "http://hibernate.sourceforge.net/hibernate-configuration-
3.0.dtd">
04:
05: <hibernate-configuration>
06:     <session-factory>
07:         <!-- Paramètres de connexion à la base -->
08:         <property name="hibernate.connection.url">
09:             jdbc:mysql://localhost/siteMarchand
10:         </property>
11:         <property name="connection.driver_class">
12:             com.mysql.jdbc.Driver
13:         </property>
14:         <property name="connection.username">nomUtilisateur
</property>
15:         <property name="connection.password">motDePasse</property>
16:         <property name="transaction.factory_class">
17:             org.hibernate.transaction.JDBCTransactionFactory
18:         </property>
19:
20:         <!-- Sélection du dialecte de génération du code SQL -->
21:         <property name="dialect">org.hibernate.dialect.
MySQLDialect</property>
22:
23:         <!-- Autres options -->
24:         <property name="hibernate.show_sql">true</property>
25:         <property name="javax.persistence.validation.mode">none
</property>
26:         <property name="current_session_context_class">thread
</property>
27:
28:         <!-- Fichier de "mapping" avec la base -->
29:         <mapping resource="Article.hbm.xml"/>
30:     </session-factory>
31: </hibernate-configuration>
```

Les lignes 8 à 18 permettent de configurer l'accès à la base de données en spécifiant l'adresse du serveur, le type de SGBD et le nom de la base au format JDBC (ligne 9), le driver JDBC à employer (ligne 12), les identifiants de connexion à la base (lignes 14 et 15), et nous déclarons que nous allons utiliser l'API de transactions d'Hibernate (lignes 16 à 18).

La ligne 21 permet de préciser le dialecte dans lequel le code SQL devra être généré (pour optimiser le code). Comme ça, nous avons la possibilité de sélectionner un SGBD et de générer le code pour un autre SGBD. C'est très pertinent... Les lignes 24 à 26 définissent trois options : l'affichage des requêtes SQL en ligne 24, la désactivation de

la validation automatique en ligne 25, et nous définissons le mode de récupération de la session courante en ligne 26. Enfin, la ligne 29 indique le fichier contenant les informations permettant de relier la table **Article** avec l'objet **Article.**

Le second fichier est celui que nous venons d'indiquer dans le fichier de configuration d'Hibernate : **Article.hbm.xml**. Il s'agit donc d'un fichier dit de « mapping » en XML. Voici le code pour la table **Article** :

```
01: <!DOCTYPE hibernate-mapping PUBLIC
02:  "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
03:  "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
04:
05: <hibernate-mapping>
06:     <class name="Article" table="Article">
07:         <id name="idArticle" type="int" column="idArticle">
08:             <generator class="assigned" />
09:         </id>
10:         <property name="designation" type="string" column="designation" />
11:         <property name="puHT" type="double" column="puHT" />
12:     </class>
13: </hibernate-mapping>
```

Ce fichier est assez simple à comprendre : nous définissons une classe **Article** qui sera liée à la table de même nom (ligne 6). Nous indiquons ensuite les différents champs de cette table : le champ **idArticle** dans les lignes 7 à 9, puis les champs **designation** et **puHT** dans les lignes 10 et 11. Ici, pour une meilleure lisibilité, les noms des objets (attributs **name**) sont les mêmes que les noms des champs dans la base (attributs **column**). Si vous le souhaitez, vous pourrez bien sûr indiquer des couples **name** / **column** différents.

Nous pouvons maintenant passer au troisième fichier, la définition de la classe **Article** dans **Article.java** :

```
01: public class Article implements java.io.Serializable
02: {
03:     private int idArticle;
04:     private String designation;
05:     private double puHT;
06:
07:     // --- Accesseurs et modifieurs ---
08:
09:     public int getIdArticle()
10:     {
11:         return this.idArticle;
12:     }
13:
14:     public void setIdArticle(int idArticle)
15:     {
16:         this.idArticle = idArticle;
17:     }
18:
19:     public String getDesignation()
20:     {
21:         return this.designation;
22:     }
23:
24:     public void setDesignation(String desc)
25:     {
26:         this.designation = desc;
27:     }
28:
29:     public double getPuHT()
30:     {
31:         return this.puHT;
32:     }
33:
34:     public void setPuHT(double puHT)
35:     {
36:         this.puHT = puHT;
37:     }
38:
39:
40:     // --- Constructeurs ---
41:
42:     public Article() {}
43:
44:     public Article(int idArticle,
String designation, double puHT)
45:     {
46:         this.idArticle = idArticle;
47:         this.
setDesignation(designation);
48:         this.setPuHT(puHT);
49:     }
50:
51:
52:     // --- Méthode toString ---
53:
54:     public String toString()
55:     {
56:         return "  - " + this.
getDesignation();
57:     }
58: }
```

On est ici sur du Java de base : création d'une classe avec définition des attributs qui correspondent aux champs de la table (lignes 3 à 5), mise en place de l'encapsulation avec la définition des méthodes d'accès et de modification dans les lignes 7 à 37, définition des constructeurs (lignes 42 à 49) et définition de la méthode **toString()** pour utiliser une instance de l'objet dans un contexte de chaîne de caractères.

Notez seulement qu'il faut bien définir pour chacun des attributs une méthode d'accès et une méthode de modification et que pour un attribut **monAttribut**, ces méthodes s'appelleront respectivement

getMonAttribut() et **setMonAttribut()**. Respectez bien la notation CamelCase si vous ne voulez pas avoir de message d'erreur ! Hibernate va rechercher ces noms et s'ils sont absents, rien ne pourra fonctionner.

L'avant-dernier fichier est la définition d'une classe qui va permettre la connexion avec le SGBD d'après les informations fournies par le fichier **hibernate.cfg.xml**. Ce fichier se nommera **HibernateUtil.java** :

```
01: import org.hibernate.SessionFactory;
02: import org.hibernate.cfg.Configuration;
03:
04: public class HibernateUtil
05: {
06:     private static final SessionFactory sessionFactory;
07:
08:     static
09:     {
10:         try
11:         {
12:             // Création de l'objet SessionFactory à partir de
hibernate.cfg.xml
13:             sessionFactory = new Configuration().configure().
buildSessionFactory();
14:         }
15:         catch (Throwable ex)
16:         {
17:             // Affichage de l'exception
18:             System.err.println("Initial SessionFactory
creation failed." + ex);
19:             throw new ExceptionInInitializerError(ex);
20:         }
21:     }
22:
23:     public static SessionFactory getSessionFactory()
24:     {
25:         return sessionFactory;
26:     }
27: }
```

Pour définir une session (un accès à la base), nous utiliserons la méthode **getSessionFactory()** définie dans les lignes 23 à 26 qui, grâce à la déclaration de la variable **sessionFactory** en tant que **static**, ouvre une nouvelle session ou nous retourne la session déjà ouverte. Les lignes 15 à 20 permettent de récupérer les exceptions pour afficher un message d'erreur si la connexion échoue.

La mise en place d'Hibernate pour notre table **Article** est terminée ! Nous pouvons écrire le code de notre programme principal **TestHibernate.java** :

```
01: import java.util.Iterator;
02: import org.hibernate.Session;
03: import org.hibernate.Query;
04:
05: public class TestHibernate
06: {
07:     public static void main(String[] args)
08:     {
09:         Session session = HibernateUtil.getSessionFactory().
getCurrentSession();
10:         Query q;
11:         Iterator results;
12:
13:         // Création de la requête
14:         session.beginTransaction();
15:         q = session.createQuery("from Article where puHT < ?");
16:         q.setDouble(0, 10000.);
17:
18:         // Parcours des résultats et affichage
19:         results = q.iterate();
20:         System.out.println("Résultats :");
21:         while (results.hasNext())
22:         {
23:             Article a = (Article) results.next();
24:             System.out.println(a);
25:         }
26:     }
27: }
```

La connexion à la base se fait par l'ouverture de la session en ligne 9. Nous indiquons le début d'une transaction (ligne 14), puis nous créons une requête préparée sur le même modèle que celui vu précédemment avec JDBC... à la différence près qu'ici, la numérotation des valeurs à insérer commence à 0 (lignes 15 et 16). Nous récupérons ensuite les résultats dans un itérateur **results** (ligne 19), que nous parcourons dans les lignes 21 à 25 pour afficher la liste des articles grâce à la méthode **toString()**.

Conclusion

L'utilisation d'une base de données depuis Java nécessite de faire attention à de nombreux détails de configuration, même si JDBC permet d'unifier le code permettant d'accéder à différents SGBD. En ce qui concerne les ORM qui sont censés simplifier la vie du développeur, nous avons pu voir que ce n'était pas vraiment le cas avec Java. Bien sûr, si Java est votre langage de prédilection, vous n'allez pas apprendre un nouveau langage seulement pour accéder à des bases de données ! Il faudra alors vous contenter de JDBC ou passer de longues heures avec Hibernate... ■

PASSEZ À LINUX !

DISPONIBLE DÈS LE 28 JUIN

Comment installer, supprimer des programmes ?

Comment transférer sa musique sur un baladeur MP3 ?

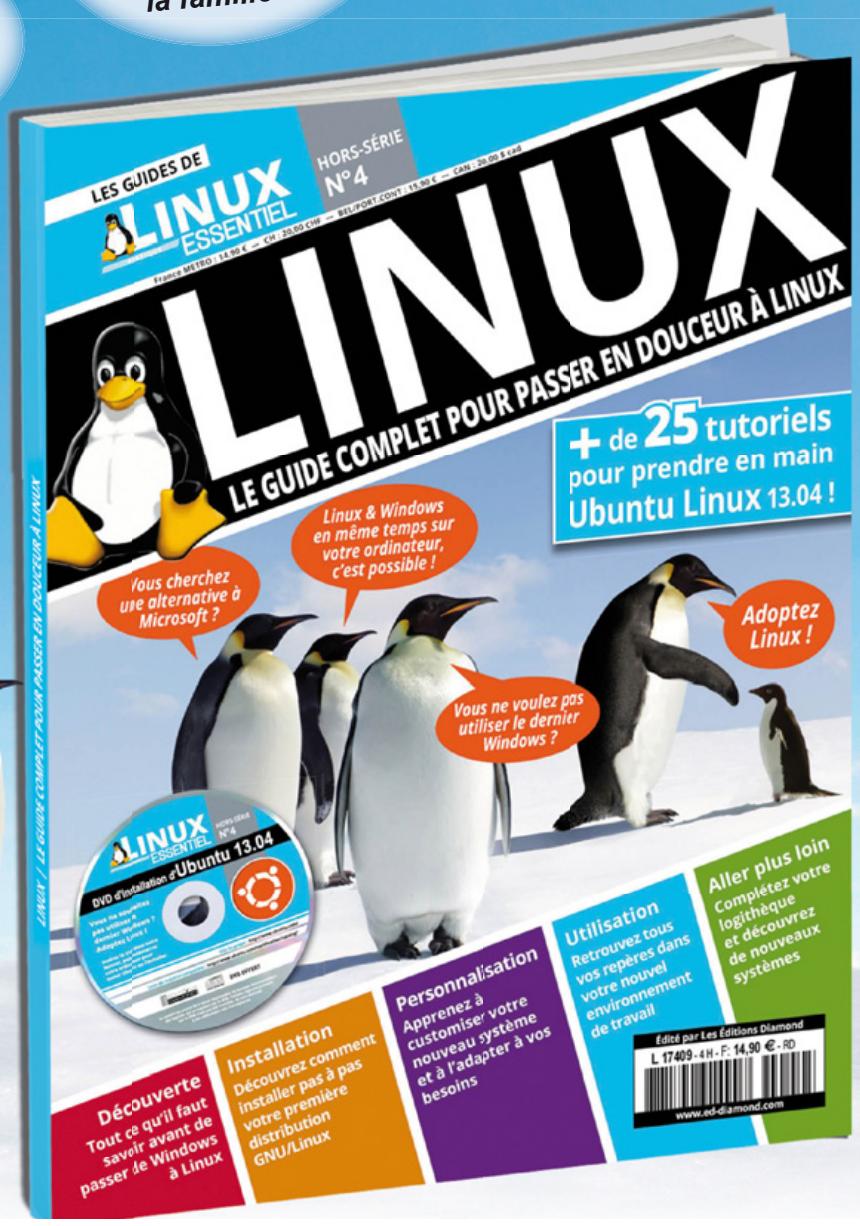
Comment créer des comptes utilisateurs pour chaque membre de la famille ?

Comment mettre à jour son système ?

Comment lire ses vidéos ?

Comment personnaliser son bureau ?

Comment stocker et synchroniser ses données en ligne ?



LINUX ESSENTIEL HORS-SÉRIE N°4

DISPONIBLE DÈS LE 28 JUIN CHEZ VOTRE MARCHAND DE JOURNAUX ET SUR : ed-diamond.com

RETROUVEZ TOUS NOS TITRES EN NUMÉRIQUE !

GNU LINUX MAGAZINE / FRANCE

MISC Multi-System & Internet Security Cookbook

GNU LINUX PRATIQUE DÉCOUVRIR, COMPRENDRE ET UTILISER LINUX

LINUX ESSENTIEL

Rendez-vous sur : numerique.ed-diamond.com

ABONNEMENTS PDF

+

REDÉCOUVREZ LES ANCIENS NUMÉROS DE TOUS NOS MAGAZINES

Abonnez-vous, achetez le magazine en PDF et feuilletez-le sur votre ordinateur, votre tablette ou votre smartphone !