Christian Soutou

Mis à jour avec MySQL 8.0

Programmer avec MySQL

SQL • Transactions • PHP • Java • Optimisations

Avec 40 exercices corrigés

5e édition

EYROLLES

Résumé

Apprendre SQL par l'exemple

Particulièrement destiné aux débutants, cet ouvrage permet de découvrir tous les aspects de la programmation SQL (création de tables, évolution, mises à jour et extractions) par le biais du système de gestion de bases de données MySQL. Les concepts du langage procédural de MySQL y sont décrits avec précision : variables, structure de contrôle, interactions avec la base, sous-programmes, curseurs, transactions, gestion des exceptions, déclencheurs, SQL dynamique… L'auteur explique en outre comment exploiter une base MySQL (connexion et transactions) en programmant avec Java (JDBC) ou PHP. Chaque notion importante du livre est introduite à l'aide d'exemples simples et chaque chapitre se clôt par une série d'exercices, avec corrigés disponibles en ligne, qui permettront au lecteur de tester ses connaissances.

Une nouvelle édition mise à jour avec MySQL 8.0

Cette cinquième édition inclut les fonctionnalités de la version 8.0 de MySQL, notamment la gestion des espaces de stockage, les fonctions SQL pour JSON et la récursivité avec les CTE. L'optimisation des requêtes est également détaillée, en particulier le fonctionnement de l'optimiseur, l'utilisation des statistiques et les plans d'exécution.

À qui s'adresse ce livre ?

- À tous ceux qui souhaitent s'initier à MySQL
- Aux développeurs Java et PHP

Sur www.editions-eyrolles.com/dl/0067379

- Téléchargez le code source des exemples et le corrigé des exercices
- Consultez les mises à jour et les compléments
- Dialoguez avec l'auteur

Au sommaire

Partie I : Les bases de SQL. Définition et manipulation des données. Évolution d'un schéma. Interrogation et contrôle des données. Partie II : Programmation procédurale. Bases du langage de programmation. Programmation avancée. Partie III : Langages et outils. Utilisation avec Java. Utilisation avec PHP. Optimisations.

Biographie auteur

Maître de conférences rattaché au département Réseaux et Télécoms de l'IUT de Blagnac, **Christian Soutou** intervient en licence et master professionnels. Il est aussi consultant indépendant chez Orsys et auteur de nombreux ouvrages aux éditions Eyrolles.

www.editions-eyrolles.com

Christian Soutou

Programmer avec MySQL

SQL • Transactions • PHP • Java • Optimisations

5e édition

EYROLLES

Attention : pour lire les exemples de lignes de code, réduisez la police de votre support au maximum.

Éditions Eyrolles
61, bd Saint-Germain
75240 Paris Cedex 05
www.editions-eyrolles.com

Avant-propos

Nombre d'ouvrages traitent de MySQL ; certains ressemblent à des bottins téléphoniques ou proviennent de la traduction hasardeuse de la documentation officielle. Les survivants ne sont peut-être plus vraiment à jour.

Ce livre a été rédigé avec une volonté de concision et de progression dans la démarche ; il est illustré, par ailleurs, de nombreux exemples et figures. Bien que la source principale d'informations fût la documentation officielle de MySQL (http://dev.mysql.com/doc), l'ouvrage ne constitue pas un condensé de commandes SQL. Chaque notion importante est introduite par un exemple simple et que j'espère démonstratif. En fin de chaque chapitre, des exercices vous permettront de tester vos connaissances.

La documentation en ligne des différentes versions de MySQL (*MySQL 5.x Reference Manual*) représente une dizaine de mégaoctets au format HTML. Tous les concepts et commandes s'y trouvant ne pourraient pas être ici décemment expliqués, sauf peut-être si cet ouvrage ressemblait à un annuaire. J'ai tenté d'en extraire seulement les aspects fondamentaux sous la forme d'une synthèse. Vous n'y trouverez donc pas des considérations à propos d'aspects avancés du langage ou du serveur comme l'administration, la mise en cluster ou la réplication.

Ce livre résulte de mon expérience de l'enseignement dans le domaine des bases de données en premier, deuxième et troisième cycles universitaires dans des cursus d'informatique à vocation professionnelle (IUT, licences et masters professionnels).

Cet ouvrage s'adresse principalement aux novices désireux de découvrir SQL en programmant sous MySQL.

- Les étudiants et enseignants trouveront des exemples pédagogiques pour chaque concept abordé, ainsi que des exercices thématiques.
- Les développeurs PHP ou Java découvriront des moyens de stocker leurs données.

Guide de lecture

Ce livre s'organise autour de trois parties distinctes mais complémentaires. La première intéressera le lecteur débutant en la matière, car elle concerne les instructions SQL et les notions de base de MySQL. La deuxième partie décrit la programmation avec le langage procédural de MySQL. La troisième partie attirera l'attention des programmeurs qui envisagent d'utiliser MySQL à l'aide d'outils natifs, ou tout en programmant avec des langages évolués ou via des interfaces Web (PHP ou Java). Enfin des aspects plus avancés de SQL sont abordés comme l'optimisation des requêtes.

Première partie : SQL de base

Cette partie présente les différents aspects du langage SQL de MySQL, en étudiant en détail les instructions de base. À partir d'exemples, j'explique notamment comment déclarer, manipuler, faire évoluer et interroger des tables avec leurs différentes caractéristiques et leurs éléments associés (contraintes, index, vues, séquences). Nous étudions aussi SQL dans un contexte multi-utilisateur (droits d'accès), et au niveau du dictionnaire de données.

Deuxième partie : programmation procédurale

Cette partie décrit les caractéristiques du langage procédural de MySQL. Le chapitre 6 traite des éléments de base (structure d'un programme, variables, structures de contrôle, interactions avec la base et transactions). Le chapitre 7 traite des sous-programmes, des curseurs, de la mise en œuvre d'exceptions, de déclencheurs et l'utilisation du SQL dynamique. La gestion des documents XML et JSON est également décrite.

Troisième partie : langages et outils

Cette partie intéressera les programmeurs qui envisagent d'exploiter une base MySQL en utilisant un langage de programmation. Le chapitre 8 détaille l'API JDBC qui permet de manipuler une base MySQL par l'intermédiaire d'un programme Java. Le chapitre 9 décrit les principales fonctions de l'API *mysqli* qui permet d'interfacer un programme PHP avec une base MySQL.

Le chapitre 10 est enfin consacré à l'optimisation des requêtes et des schémas. Plusieurs aspects sont étudiés : le fonctionnement de l'optimiseur, l'utilisation de statistiques et les plans d'exécution. Enfin, différents mécanismes permettant d'optimiser les traitements sont présentés : contraintes, index, tables temporaires, partitionnement et dénormalisation.

Annexes

Les annexes contiennent une bibliographie, des adresses Web et un index complet.

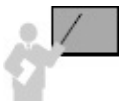
Convention d'écriture

La police `courrier` est utilisée pour souligner les instructions SQL, noms de types, tables, contraintes, etc. (ex : `SELECT nom FROM Pilote`).

Les majuscules sont employées pour les directives SQL, et les minuscules pour les autres éléments. Les noms des tables, index, vues, fonctions, procédures... sont précédés d'une majuscule (exemple : la table `CompagnieAerienne` contient la colonne `nomComp`).

Les termes de MySQL (bien souvent traduits littéralement de l'anglais) sont notés en italique, exemple : *trigger*, *table*, *column*, etc.

Dans une instruction SQL, les symboles { } désignent une liste d'éléments, et le symbole « | » un choix (exemple `CREATE { TABLE | VIEW }`. Les symboles « [» et «] » précisent le caractère optionnel d'une directive au sein d'une commande (exemple : `CREATE [UNIQUE|FULLTEXT|SPATIAL] INDEX index_name [USING index_type] ON table_name (index_col_name,...)`).



Ce sigle introduit une définition, un concept ou une remarque importante. Il apparaît soit dans une partie théorique soit dans une partie technique pour souligner des instructions importantes ou la marche à suivre avec SQL.

Ce sigle annonce soit une impossibilité de mise en œuvre d'un concept soit une mise en garde. Il est principalement utilisé dans la partie consacrée à SQL.

J'en profite pour faire passer le message suivant : si vous travaillez en version 4 de MySQL, certaines instructions décrites dans ce livre ne fonctionneront pas. Cet ouvrage n'est pas un guide de référence ! Vous trouverez sur le Web des ressources pour connaître la compatibilité de telle ou telle fonction SQL.

Ce sigle indique à partir de quelle version du serveur MySQL la fonctionnalité (commande SQL ou mécanisme de programmation) est opérationnelle. Les versions 5.1 à 8.0 sont prises en compte.

Ce sigle signale une astuce ou un conseil personnel.

Contact avec l'auteur

Si vous avez des remarques à formuler sur le contenu de cet ouvrage, n'hésitez pas à m'écrire à l'adresse christian.soutou@gmail.com, mais seules les remarques relatives à l'ouvrage devraient trouver une réponse.

Par ailleurs, un site d'accompagnement de l'ouvrage (*errata*, corrigés des exercices, source des exemples et compléments) est disponible, accessible via www.editions-eyrolles.com sur la fiche de l'ouvrage.

Table des matières

Introduction

Dans cette introduction, nous présentons, tout d'abord, le cadre général dans lequel cet ouvrage se positionne (SQL, le modèle de données et l'offre MySQL). Nous décrivons, pour finir, la procédure d'installation de MySQL sous Windows et l'utilisation de l'interface de commande en ligne pour que vous puissiez programmer en SQL dès le chapitre 1.

SQL, une norme, un succès

C'est IBM, *à tout seigneur tout honneur*, qui, avec System-R, a implanté le modèle relationnel au travers du langage SEQUEL (*Stuctured English as QUEry Language*), rebaptisé par la suite SQL (*Structured Query Language*).

La première norme (SQL1) date de 1987. Elle était le résultat de compromis entre constructeurs, mais elle était fortement influencée par le dialecte d'IBM. SQL2 a été normalisée en 1992. Elle définit quatre niveaux de conformité : le niveau d'entrée (*entry level*), les niveaux intermédiaires (*transitional* et *intermediate levels*) et le niveau supérieur (*full level*). Les langages SQL des principaux éditeurs sont tous conformes au premier niveau et ont beaucoup de caractéristiques relevant des niveaux supérieurs. Depuis 1999, la norme est appelée SQL3. Elle comporte de nombreuses parties (concepts objets, entrepôts de données, séries temporelles, accès à des sources non SQL, réplication des données, etc.).

Le succès que connaissent les éditeurs de SGBD relationnels a plusieurs origines et repose notamment sur SQL :

- Le langage est une norme depuis 1986, qui s'enrichit au fil du temps.
- SQL peut s'interfacer avec des langages de troisième génération comme C ou Cobol, mais aussi avec des langages plus évolués comme C++, Java ou C#. Certains considèrent ainsi que le langage SQL n'est pas assez complet (le dialogue entre la base et l'interface n'est pas direct), et la littérature parle de « défaut d'impédance » (*impedance mismatch*).

- Les SGBD rendent indépendants programmes et données (la modification d'une structure de données n'entraîne pas forcément une importante refonte des programmes d'application).
- Ces systèmes sont bien adaptés aux grandes applications informatiques de gestion (architectures type client-serveur et Internet) et ont acquis une maturité sur le plan de la fiabilité et des performances.
- Ils intègrent des outils de développement comme les précompilateurs, les générateurs de code, d'états, de formulaires.
- Ils offrent la possibilité de stocker des informations non structurées (comme le texte, l'image, etc.) dans des champs appelés LOB (*Large Object Binary*).

Nous étudierons les principales instructions SQL de MySQL qui sont classifiées dans le tableau suivant :

Tableau I-1 Classification des ordres SQL

Ordres SQL	Aspect du langage
`CREATE – ALTER – DROP - COMMENT – RENAME – TRUNCATE - GRANT - REVOKE`	Définition des données (DDL : *Data Definition Language*)
`SELECT - INSERT – UPDATE – DELETE – LOCK TABLE`	Manipulation des données (DML : *Data Manipulation Language*)
`COMMIT – ROLLBACK – SAVEPOINT – SET TRANSACTION`	Contrôle des transactions (TCL : *Transaction Control Statements*)

Modèle de données

Le modèle de données relationnelles repose sur une théorie rigoureuse bien qu'adoptant des principes simples. La table relationnelle (*relational table*) est la structure de données de base qui contient des enregistrements appelés aussi « lignes » (*rows*). Une table est composée de colonnes (*columns*) qui décrivent les enregistrements.

Tables et données

Considérons la figure suivante qui présente deux tables relationnelles permettant de stocker des compagnies, des pilotes et le fait qu'un pilote soit embauché par une compagnie :

Figure I-1 *Deux tables*

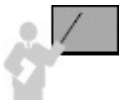
Compagnie

comp	nrue	rue	ville	nomComp
AF	124	Port Royal	Paris	Air France
SING	7	Camparols	Singapour	Singapore AL

Pilote

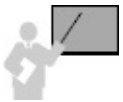
brevet	nom_pil	pseudo	nbHVol	*compa*
PL-1	Ferrage	Shark	450	AF
PL-2	Tisseyre	Ninja	900	AF
PL-3	Guilbaud	Thai	1000	SING

Les clés



La clé primaire (*primary key*) d'une table est l'ensemble minimal de colonnes qui permet d'identifier de manière unique chaque enregistrement.

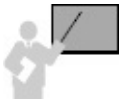
Dans la figure précédente, les colonnes « clés primaires » sont notées en gras. La colonne `comp` représente le code de la compagnie et la colonne `brevet` décrit le numéro du brevet.



Une clé est dite « candidate » (*candidate key*) si elle peut se substituer à la clé primaire à tout instant. Une table peut contenir plusieurs clés candidates ou aucune.

Dans notre exemple, les colonnes `nomComp` et `pseudo` peuvent être des clés

candidates si on suppose qu'aucun homonyme n'est permis.



Une clé étrangère (*foreign key*) référence dans la majorité des cas une clé primaire d'une autre table (sinon une clé candidate sur laquelle un index unique aura été défini). Une clé étrangère est composée d'une ou de plusieurs colonnes. Une table peut contenir plusieurs clés étrangères ou aucune.

Dans notre exemple, la colonne `compa` (notée en italique dans la figure) est une clé étrangère, car elle permet de référencer un enregistrement unique de la table `Compagnie` via la clé primaire `comp`.

Le modèle relationnel est ainsi fondamentalement basé sur les valeurs. Les associations entre tables sont toujours binaires et assurées par les clés étrangères. Les théoriciens considèrent celles-ci comme des pointeurs logiques. Les clés primaires et étrangères seront définies dans les tables en SQL à l'aide de contraintes.

MySQL

Le SGBD MySQL a été développé en C et C++ par l'équipe suédoise TcX, dans l'objectif d'améliorer le logiciel mSQL. La première version est apparue en mai 1995, elle fut distribuée par la société MySQL AB (Uppsala, Suède) fondée par David Axmark, Allan Larsson et Michael Widenius. Voici le modeste slogan affiché sur le site http://www.mysql.com : *The world's most popular open source database* (« La base de données open source la plus populaire au monde »).

Depuis 1999, par analogie avec les systèmes d'exploitation, MySQL connaît le succès de Linux. Téléchargée plus d'un million de fois par mois, la version *production* doit sa popularité à son caractère *open source*, ses fonctionnalités de plus en plus riches, ses performances, son ouverture à tous les principaux langages du marché, son fonctionnement sur les systèmes les plus courants et sa facilité d'utilisation pour des applications Web de taille moyenne.

Les produits

Les produits de la société MySQL sont les suivants :

- *MySQL Enterprise*, qui inclut *MySQL Enterprise Server* (le SGBD version complète), *MySQL Enterprise Monitor* (la console d'administration) et *MySQL Production Support* (le service de support) ;
- *MySQL Cluster*, qui implémente la solution de haute disponibilité (architecture en *cluster*) ;
- *MySQL Embedded Server* (SGBD seul), supporté par plus de 20 plates-formes, de Microsoft Windows à Mac OS X en passant par Linux, Sun Solaris, HP-UX, IBM AIX, Novell Netware, etc. ;
- *MySQL Connectors*, pilotes (*drivers*) permettant un accès à tout programme. Citons principalement ADO.NET (*Connector/NET*), ODBC (*Connector/ODBC*), JDBC (*Connector/J*), C++ (*Connector/C++*), C (*Connector/C*), *MySQL Connector for OpenOffice.org*, pilotes pour PHP, Perl, Python et Ruby ;
- *MySQL Workbench*, outil graphique pour les DBA, développeurs et concepteurs.
- *MySQL Fabric*, qui permet d'administrer des serveurs en cluster en basculant par exemple, un serveur esclave pour devenir primaire suite à un incident sur le serveur principal.

Licences

Deux types de licences sont proposés par MySQL : commerciale et GPL (depuis la version 3.23.19 en juin 2000). Dans le cadre d'un développement d'application entièrement sous licence GPL, MySQL est gratuit. Il en va de même s'il n'est pas copié, modifié, distribué ou employé pour une utilisation en combinaison avec un serveur Web (si vous développez l'application Web vous-même).

Dans tous les autres cas, il est nécessaire d'obtenir une licence commerciale. Par exemple, si vous incluez un serveur MySQL ou des pilotes MySQL dans une application qui n'est pas *open source*.

Les versions

Le tableau suivant présente l'historique des versions de MySQL. Le rythme des mises à jour majeures (*Generally Available*, aussi appelées GA) est d'environ une tous les 18 à 24 mois. Des versions intermédiaires (*Milestone* puis *Release Candidate*) apparaissent régulièrement entre deux versions de production.

Tableau I-2 Dates importantes pour MySQL

Année – version	Caractéristiques principales
1999 – 3.23.*x*	Réplication – Recherches textuelles – Transactions – Intégrité référentielle pour le moteur InnoDB (2002 – 3.23.44)
2001 – 4.0.*x*	Cache de requêtes – Sécurisation SSL
2003 – 4.1.*x*	Support de Unicode – Données géographiques – SQL dynamique
2005 – 5.0.*x*	Vues – Curseurs – Procédures cataloguées – Déclencheurs – Dictionnaire des données – Transactions distribuées (XA)
2008 – 5.1.*x*	Jointures externes – Planificateur d'événements – Partitionnement – Réplication au niveau ligne – Tables *server log*
2010 – 5.5	Nouveaux algorithmes de requêtes – Optimisations des sous-requêtes – Montée en charge accrue
2012 – 5.6	Évolutions de l'optimiseur et de la réplication – NoSQL pour InnoDB – Évolutions SIG - Conformité IPv6
2015 – 5.7	Évolutions de l'optimiseur, de la sécurité – Prise en compte de JSON – Mode Document (NoSQL) – X Protocol – Évolutions SIG
2017 – 8.0	Dictionnaire des données transactionnel – Rôles – Index invisibles – Abandon de MySAM

Vous trouverez toutes les nouvelles fonctionnalités à l'adresse suivante : http://dev.mysql.com/doc/relnotes/mysql/x.y/en/ (*x.y* étant le numéro de version, par exemple 5.7).

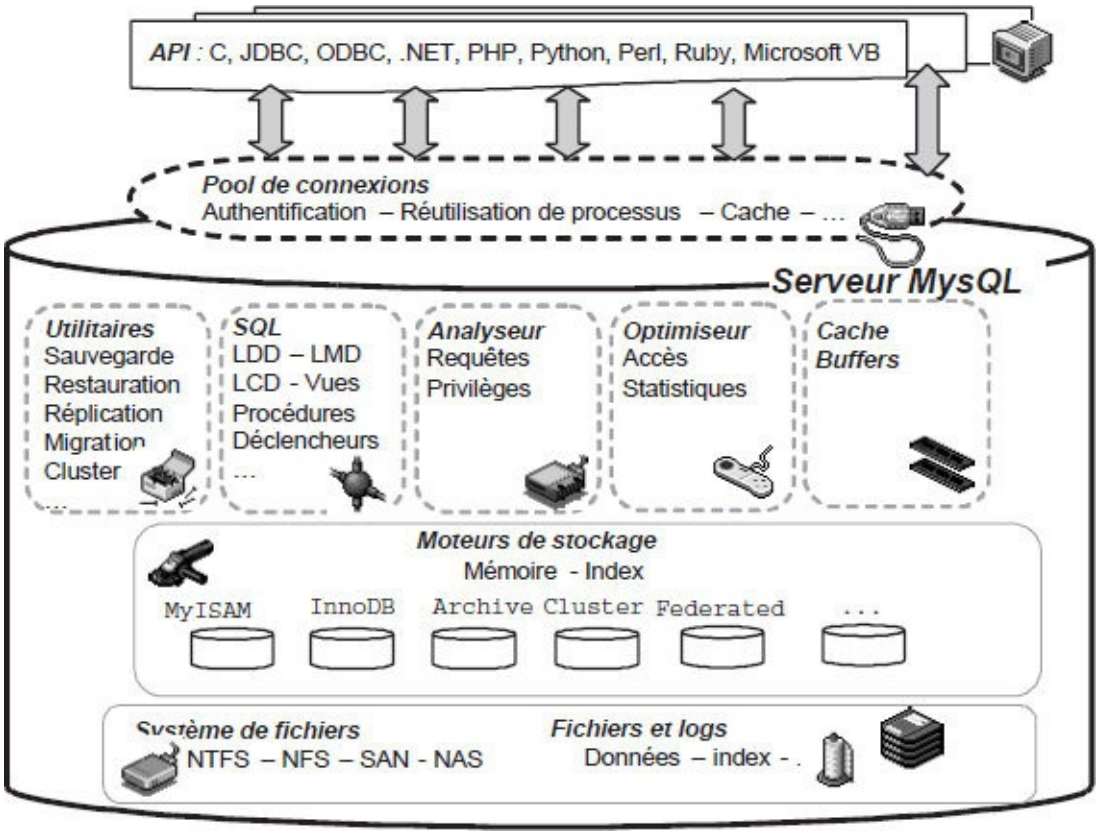
Architecture

La figure suivante (je remercie au passage le site http://www.iconarchive.com) présente la majeure partie des fonctionnalités du serveur de données (SGBD).

Les API permettent d'intégrer SQL dans des programmes de différents langages. Le langage SQL sera utilisé par tous ceux (manuellement ou par un outil) travaillant sur la base de données (administrateur, développeur, utilisateur). Le langage procédural de MySQL permet d'incorporer nativement tout ordre SQL dans un programme.

Concrètement, une fois téléchargé et installé, vous avez accès à un SGBD, un client en mode texte (interface de commande). Les pilotes ODBC, JDBC, API pour les langages C et C++, et les outils d'administration seront à installer par la suite. Si vous désirez utiliser MySQL dans le cadre d'un site Web, il existe des paquetages incluant notamment Apache et PHP (WAMP ou LAMP respectivement pour Windows et Linux). Le chapitre 9 décrit une procédure manuelle pour interconnecter ces systèmes.

***Figure I-2** Offre MySQL*



Moteurs de stockage

La particularité du SGBD MySQL est de pouvoir proposer différents moteurs de stockage. Le choix de tel ou tel moteur dépendra de la façon dont on veut stocker et traiter les données de chaque table (voir le chapitre 1).

Les principaux paramètres à prendre en considération concernent les capacités de stockage, les transactions, les modes de verrouillage, la gestion des index, des caches, des données textuelles, géographiques, compressées ou crytées, etc. Plusieurs moteurs sont proposés par la société MySQL AB, d'autres sont issus de communautés ou d'éditeurs indépendants. Parmi les moteurs natifs les plus utilisés, citons :

- MyISAM : moteur par défaut. Ne supporte pas les transactions mais possède des fonctionnalités de recherche de texte.
- InnoDB : sans doute le plus utilisé de nos jours. Supporte le mode transactionnel (verrouillage niveau ligne, *commit* et *rollback*) et les contraintes référentielles (clés étrangères).
- MEMORY (anciennenemt HEAP) : stockage des données et index en RAM. Convient à des données non persistantes.
- ARCHIVE : stockage des données sous une forme compressée (seuls les `SELECT` et `INSERT` sont possibles). Modèle idéal pour archiver des données.
- CSV (*Comma Separated Value*) : stockage des données sous forme de fichiers texte dans lesquels les valeurs sont séparées par une virgule.
- FEDERATED : convient pour les architectures réparties (plusieurs serveurs).
- NDB (*Network DataBase*) : convient pour les architectures en cluster.

Oracle devient propriétaire

Contrairement à la rumeur qui courait début 2007, MySQL n'est pas entré en Bourse. Il a été racheté pour un milliard de dollars en janvier 2008 par Sun Microsystems, déjà propriétaire de Java. Sun se positionne ainsi sur un segment où il était encore absent jusque là, aux côtés d'Oracle, d'IBM et de Microsoft.

Craignant l'achat de Sun par IBM et redoutant HP dans le haut de gamme Unix, Oracle se repositionne dans le hardware et sur le marché des services pour

datacenters en avril 2009, en achetant Sun. Ce sont aussi les langages Java et le système d'exploitation Solaris qui ont pesé dans la balance. En effet, c'est sur Solaris, et non sur Linux, que sont déployés le plus grand nombre de serveurs Oracle.

Il faudra attendre novembre 2009 pour que la Commission européenne confirme son refus de la fusion entre Oracle et Sun, suspectant que le rachat de MySQL aboutisse à une situation de quasi monopole sur le marché des SGDB. En décembre 2009, avec le soutien de quelques cinquante-neuf sénateurs américains, Oracle publie dix engagements concernant toutes les zones géographiques et pour une durée de cinq ans :

1. Assurer aux utilisateurs le choix de leur moteur (*MySQL's Pluggable Storage Engine Architecture*).
2. Ne pas changer les clauses d'utilisation d'une manière préjudiciable à un fournisseur tiers.
3. Poursuivre les accords commerciaux contractés par Sun.
4. Garder MySQL sous licence GPL.
5. Ne pas imposer un support des services d'Oracle aux clients du SGBD.
6. Augmenter les ressources allouées à la R&D de MySQL.
7. Créer un comité d'utilisateurs pour, dans les six mois, étudier les retours et priorités de développement de MySQL.
8. Créer ce même comité pour les fournisseurs de solutions incluant MySQL.
9. Continuer d'éditer, mettre à jour et distribuer gratuitement le manuel d'utilisation du SGBD.
10. Laisser aux utilisateurs le choix de la société qui assurera le support de MySQL.

Considérant d'une part ces engagements, et d'autre part l'existence de concurrents (notamment IBM, Microsoft et PostgreSQL dans le monde de l'open source), la Commission européenne avalise la fusion fin janvier 2010 pour un montant de 7,4 milliards de dollars.

Notion de database [schéma]

MySQL appelle *database* un regroupement logique d'objets (tables, index, vues, déclencheurs, procédures cataloguées, etc.) pouvant être stockés à

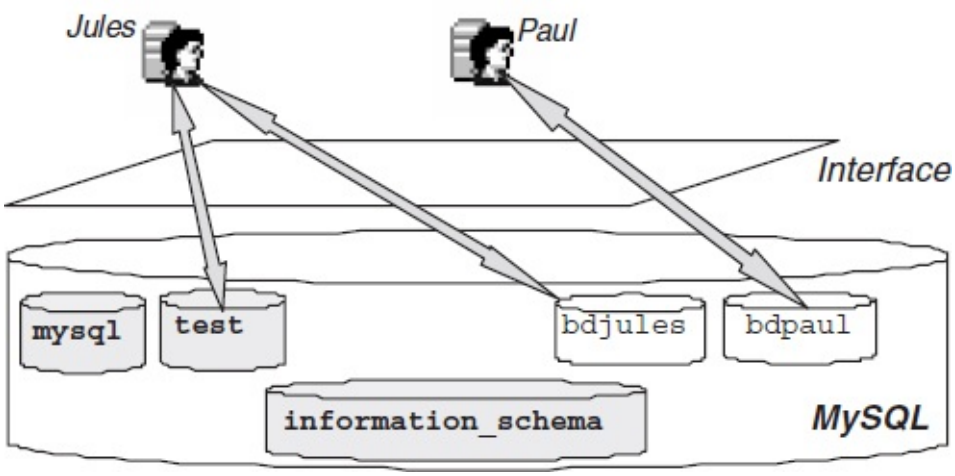
différents endroits de l'espace disque. Je ferai donc souvent référence au terme « base de données » pour parler de cette notion.

On peut aussi assimiler ce concept à la notion de schéma, pour ceux qui connaissent Oracle. Là où MySQL et d'autres SGBD diffèrent, c'est sur la notion d'utilisateur (*user*) :

- Pour tous, un utilisateur sera associé à un mot de passe pour pouvoir se connecter et manipuler des tables (s'il en a le droit, bien sûr).
- Pour MySQL, il n'y a pas de notion d'appartenance d'un objet (table, index, etc.) à un utilisateur. Un objet appartient à son schéma (*database*). Ainsi, deux utilisateurs distincts (`Jules` et `Paul`) se connectant sur la même base (*database*) ne pourront pas créer chacun une table ayant pout nom `Compagnie`. S'ils doivent le faire, ce sera dans deux bases différentes (`bdjules` et `bdpaul`).
- Pour Oracle ou d'autres SGBD, chaque objet appartient à un schéma (*user*). Ainsi, deux utilisateurs distincts (`Jules` et `Paul`) se connectant à la base (qui est un ensemble de schémas) pourront créer chacun une table ayant pour nom `Compagnie` (la première sera référencée `Jules.Compagnie`, la seconde `Paul.Compagnie`).

La figure suivante illustre deux utilisateurs travaillant sur différentes bases par une interface qui peut être la fenêtre de commande en ligne ou un langage de programmation via une API C, Java ou PHP. Notez l'existence de trois bases présentes initialement (`mysql`, `test` et `information_schema`) que nous détaillerons au chapitre 5.

Figure I-3 *Bases et utilisateurs MySQL*



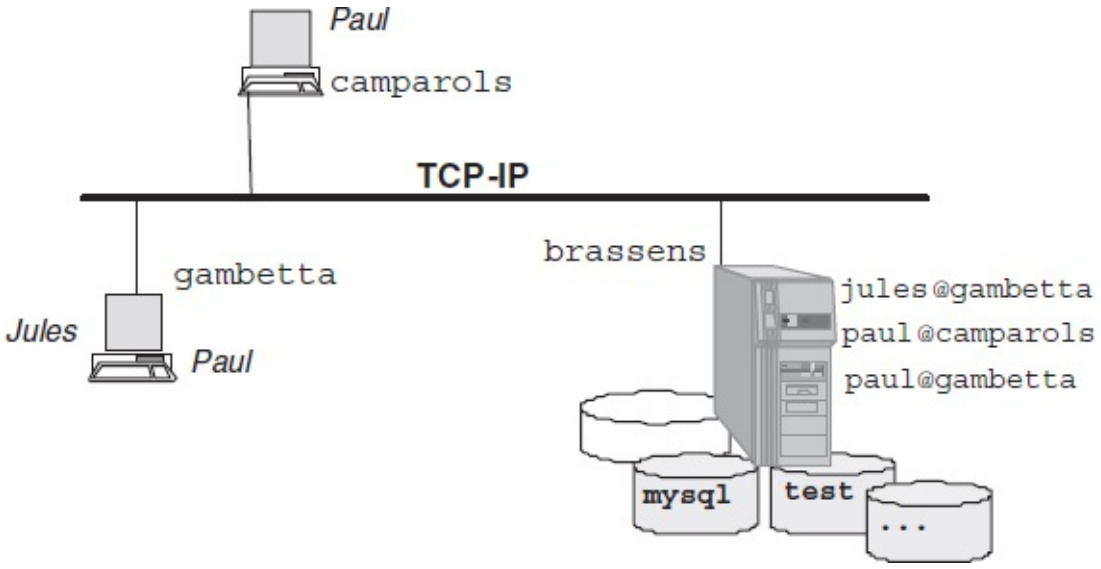
Notion d'hôte

MySQL dénomme *host* la machine hébergeant le SGBD. MySQL diffère aussi à ce niveau des autres SGBD, car il est possible de distinguer des accès d'un même utilisateur suivant qu'il se connecte à partir d'une machine ou d'une autre. La notion d'identité est basée sur le couple nom d'utilisateur MySQL (*user*) côté serveur, machine cliente.

Identités

Ainsi, l'utilisateur `Paul`, se connectant depuis la machine `camparols`, peut ne pas être le même que celui se connectant depuis la machine `gambetta`. S'il s'agit du même, il faudra, au niveau du serveur, éventuellement composer un ensemble de prérogatives équivalent pour les deux accès (voir le chapitre 5). S'il s'agit de deux personnes différentes, il faudra distinguer les attributions des différents droits. La figure suivante illustre le fait que deux utilisateurs peuvent se connecter par deux accès différents. Trois identités seront donc à créer et à administrer côté serveur.

Figure I-4 *Notion d'hôte MySQL*



Le chapitre 5 décrit la configuration d'un serveur et de clients. Reportez-vous également au site d'accompagnement de l'ouvrage qui présente quelques outils graphiques d'aide à l'administration.

Accès à MySQL

Une fois que vous aurez installé MySQL sur votre ordinateur, vous serez libre de choisir l'accès qui vous convient. Ce livre utilise essentiellement l'interface en ligne de commandes fournie avec le SGBD, mais aussi Java via JDBC, et le navigateur Web au travers de PHP.

Aspects étudiés

Nous n'étudierons pas tous les concepts d'un serveur MySQL, car certains sont assez spécifiques. Le tableau suivant indique le chapitre du livre dans lequel vous trouverez le descriptif du concept étudié.

Tableau I-3 Éléments d'une base MySQL

Éléments étudiés – Chapitre	Aspects non étudiés
Tables et index – Chapitre 1 Séquences – Chapitres 2 et 5 Vues (*views*) et utilisateurs – Chapitre 5 Déclencheurs (*triggers*) – Chapitre 7 Fonctions et procédures (*stored programs*) – Chapitre 7 Mode document (NoSQL) – Chapitre 7 Moteurs de stockage (*storage engines*) – Chapitre 10 Partitionnement – Chapitre 10	Haute disponibilité Réplication Sauvegardes et restaurations Données spatiales (GIS)

Décrivons à présent l'installation du SGBD sous Windows. À noter que les options d'installation sont identiques sous Linux.

Mise en œuvre de MySQL [sous Windows]

Si la procédure se déroule sans problème, vous n'aurez besoin que de quelques minutes pour installer MySQL. Je vous conseille toutefois de créer un point de restauration pour pouvoir revenir si besoin à la dernière bonne configuration connue.

Pour obtenir les dernières versions en cours de développement et celles de

production, rendez-vous sur le site http://www.mysql.com/downloads, puis sélectionnez MySQL Community Server.

La première étape consiste à sélectionner votre plate-forme (qui va de Windows à Mac OS en passant par différentes éditions de Linux). Pour pouvoir télécharger le logiciel et poster sur les forums officiels, vous devez vous enregistrer (si vous disposez déjà d'un compte Oracle, c'est le même). Dans le cas de Windows, le logiciel MySQL est fourni avec un fichier d'installation (*MSI Installer*) qu'il convient d'utiliser.

Installation

Après avoir accepté les termes de la licence, vous êtes invité à choisir le répertoire d'installation (par défaut `Program Files\MySQL`) et le type d'installation (par défaut *Developer*). Ensuite, une étape de mise à jour de composants et de vérification des prérequis s'opère, au cours de laquelle vous pouvez agir avant le récapitulatif des produits qui seront installés.

La configuration de base inclut :

- le mot de passe de l'utilisateur `root` qui permet d'ajouter des nouveaux utilisateurs en leur accordant des rôles prédéfinis (administrateur, concepteur, importateur, sauvegarde, etc.) ;
- un service Windows dédié à MySQL (nommé par défaut MySQLxx, qu'il sera possible d'arrêter via le panneau de configuration) ;
- le port UDP d'écoute (par défaut 3306), vous pouvez ainsi ajouter une exception à votre pare-feu (option cochée par défaut).

Suivant la version choisie, la configuration avancée vous permettra d'agir sur :

- le type de votre serveur (machine de développement, serveur ou machine dédiée) ;
- les répertoires qui contiendront les fichiers de trace ;
- L'inclusion du chemin de l'exécutable `mysql` dans la variable d'environnement (`path`) ;
- le mode comportemental du serveur par rapport à la syntaxe des instructions SQL ;
- le type de base de données (multifonction, mode transactionnel ou pas) ;

- nombre de connexions (15 par défaut) ;
- le jeu de caractères (West European pour nous).

Il est possible d'ajouter un utilisateur MySQL (autre que `root`) lors de l'installation mais ce n'est pas la peine de le faire car il vaut mieux le faire ajouter une fois l'installation terminée.

Une fois MySQL installé, et suivant la version choisie et celle de votre Windows, c'est peut-être la fête des répertoires car des arborescences de MySQL risquent d'apparaître dans les trois répertoires suivants : `C:\Program Files`, `C:\ProgramData\MySQL` (par défaut invisible dans l'explorateur) et `C:\Program Files (x86)`. Le sous-répertoire `MySQL Server x.x` indique la version de votre serveur.

Le fichier de configuration `my.ini` (qui se trouve en principe dans `C:\ProgramData\MySQL`) est stratégique : il contient un grand nombre de paramètres relatifs au serveur. Si vous devez modifier un paramètre, vous devrez redémarrer le service. Un conseil toutefois : copiez ce fichier initial pour pouvoir annuler vos tentatives de modifications en cas de refus de démarrage du service MySQL.

Vos bases de données se trouveront dans le répertoire indiqué par la variable `datadir` contenue dans le fichier `my.ini`. Il s'agit de `C:\ProgramData\MySQL\MySQL Server x.x\Data` par défaut pour Windows 7 et de `C:\Documents and Settings\All Users\Application Data\MySQL\MySQL Server x.x` pour Windows XP.

Si cela n'a pas été fait, ajoutez le chemin `C:\Program Files\MySQL\MySQL Server x.x\bin` à votre variable d'environnement `path` (système ou utilisateur). Pour XP, `Poste de travail/Propriétés/Avancé`, puis `Variables d'environnement` en choisissant la variable système `path`. Pour Vista ou Seven, le poste de travail s'appelle *Ordinateur*, ensuite vous trouverez le même processus via `Propriétés/Paramètres système avancés`. N'oubliez pas d'ajouter le point-virgule pour séparer votre nouveau chemin du dernier des différents chemins existants.

Premiers pas

Cette section va guider vos premiers pas pour travailler avec l'interface de commandes en ligne. Il s'agira de stocker les fichiers qui vous serviront à effectuer différentes actions (créations de tables, de vues ou d'utilisateurs,

insertions, modifications ou suppressions d'enregistrements, élaboration de requêtes, de procédures cataloguées, etc.).

L'interface de commande

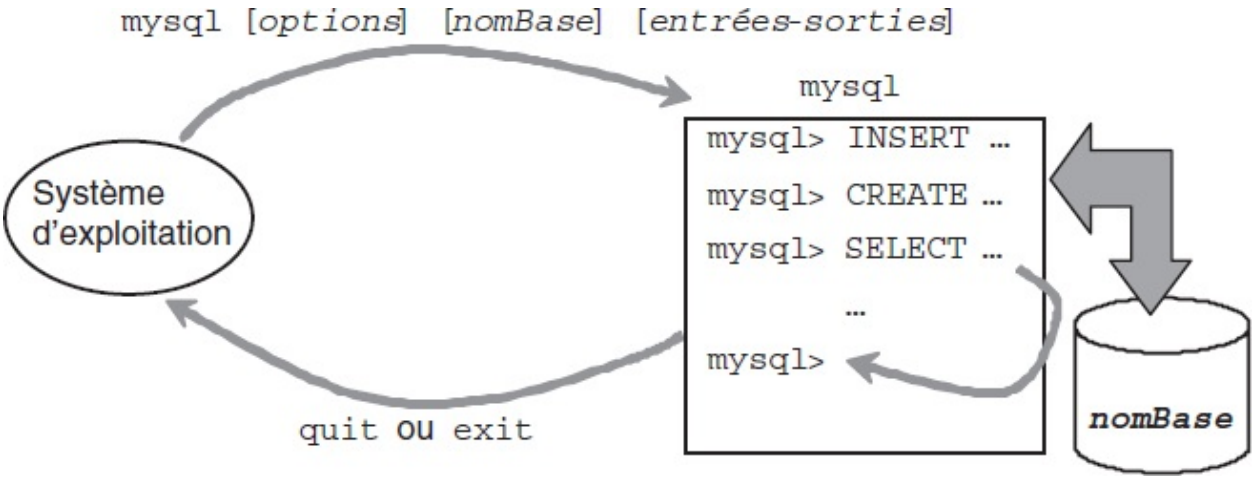
L'interface en ligne de commande se lance grâce à l'exécutable `mysql`. Cette interface ressemble à une fenêtre DOS ou `Telnet` et permet de dialoguer très simplement avec la base de données. L'utilisation peut être interactive ou en mode *batch*. Dans le premier cas (c'est le mode le plus courant), le résultat des extractions est présenté sous une forme tabulaire au format ASCII.

Vous verrez qu'il est notamment possible :

- d'exécuter des instructions SQL (créer des tables, manipuler des données, extraire des informations, etc.) ;
- de compiler des procédures cataloguées et des déclencheurs ;
- de réaliser des tâches d'administration (création d'utilisateurs, attribution de privilèges, etc.).

Le principe général de l'interface est le suivant : après une connexion locale ou distante, des instructions sont saisies et envoyées à la base qui retourne des résultats affichés dans la même fenêtre de commande.

Figure I-5 *Principe général de l'interface en ligne de commande*



N’ayez pas honte de bien maîtriser cette interface au lieu de connaître toutes les options d’un outil graphique (comme *PhpMyAdmin*, *MySQL Administrator* ou autre). Il vous sera toujours plus facile de vous adapter aux différents boutons et menus, tout en connaissant les instructions SQL, que l’inverse.

Imaginez-vous un jour à Singapour sur une machine ne disposant d’aucun outil graphique et qu’un client vous demande la réduction que vous pouvez lui faire sur la piscine intérieure d’un Airbus A380. Vous devez interroger une table sur le serveur du siège social à Blagnac et vous ne savez pas vous servir de l’interface en ligne de commande : vous n’êtes pas un informaticien !

Création d’un utilisateur

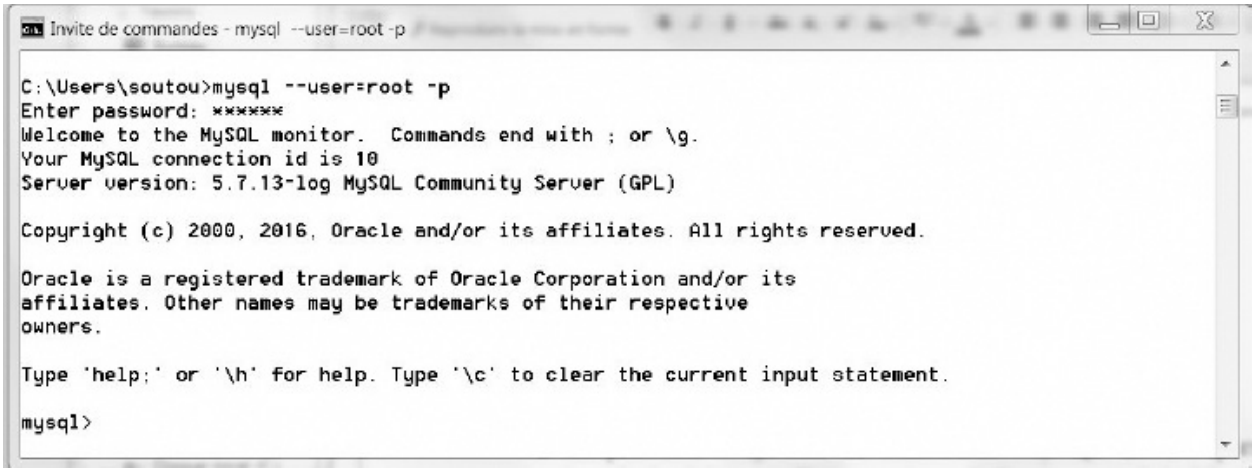
Vous allez maintenant créer un utilisateur MySQL. Pour cela, ouvrez le fichier `premierPas.sql` qui se trouve dans le répertoire `Introduction`, à l’aide du bloc-notes (ou d’un éditeur de texte de votre choix). Remplacez « util » par le nom de l’utilisateur à créer (modifiez aussi le nom de la base). Vous pouvez également changer le mot de passe si vous le souhaitez. Enregistrez ce fichier dans l’un de vos répertoires.

Connexion au serveur

Dans une fenêtre de commande Windows, Linux (ou autre), lancez l’interface en ligne de commandes en connectant l’utilisateur `root` avec le mot de passe spécifié lors de l’installation :

```
mysql --user=root -p
```

Figure I-6 *Interface en mode ligne de commande*



Une fois connecté, par copier-coller (en effectuant un clic droit dans la fenêtre de commande MySQL), exécutez une à une les différentes instructions (création de la base, de l'utilisateur, des privilèges et déconnexion de l'utilisateur `root`). Nous étudierons au chapitre 5 les notions élémentaires de droits et de sécurité. Les lignes encadrées par les signes `/*` et `*/` sont des commentaires.

Votre utilisateur (*util*) est désormais créé, il peut se connecter et il possède toutes les prérogatives sur la base (*bdutil*) pour exécuter les instructions décrites dans cet ouvrage. Pour tester votre connexion, lancez la commande suivante qui se connecte au serveur sur la base *bdutil*, sous l'utilisateur *util*.

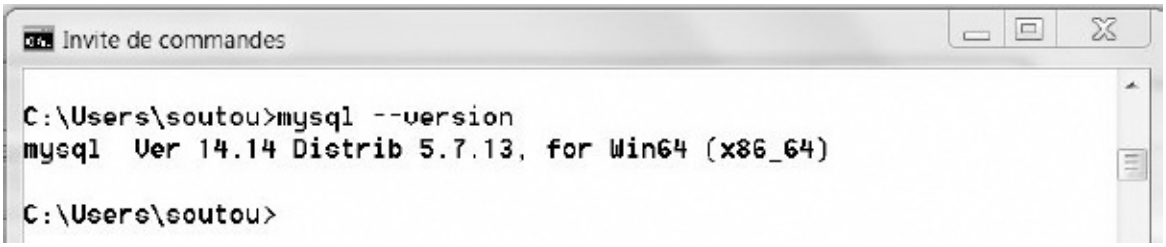
```
mysql --user=util --host=localhost -p --database=bdutil
```

Vérification de la version

Pour contrôler la version de votre serveur, exécutez la connexion-déconnexion suivante dans une fenêtre de commande (Windows ou Linux) :

```
mysql --version
```

Figure I-7 *Version du serveur MySQL*



Si vous êtes déjà connecté, la commande `SELECT VERSION();` vous renseignera de la même manière à ce propos.

Options de base

Les principales options au lancement de `mysql` sont résumées dans le tableau suivant :

Tableau I-4 Principales options de la commande `mysql`

Option	Commentaire
`--help` ou `-?`	Affiche les options disponibles, l'état des variables d'environnement et rend la main.
`--batch` ou `-B`	Toute commande SQL peut être lancée dans la fenêtre de commandes système sans pour autant afficher l'invite. Les résultats (colonnes) sont séparés par des tabulations.
`--database=`*`nomBD`* ou `–D` *`nomBD`*	Sélection de la base de données à utiliser après la connexion.
`--host=`*`nomServeur`* ou `-h` *`nomServeur`*	Désignation du serveur.
`--html` ou `-H`	Formate le résultat des extractions en HTML.
`--one-database` ou `-O`	Restreint les instructions à la base de données spécifiée initialement.
`-p`	Demande le mot de passe sans l'employer en tant que paramètre.
`--password=`*`motdePasse`*	Transmission du mot de passe de l'utilisateur à connecter. Évitez le plus possible cette option et préférez-lui la précédente.
`--prompt=`*`parametre`*	Personnalise l'invite de commande (par défaut `mysql>`).
`--silent` ou `-s`	Configure le mode silence pour réduire les messages de MySQL.
`--skip-column-names` ou `-N`	N'écrit aucun en-tête de colonne pour les résultats d'extraction.

`--table` ou `-t`	Formate le résultat des extractions en tables à en-tête de colonne (par défaut dans le mode interactif).
`--tee=`*`cheminNomFichier`*	Copie la trace de toute la session dans le fichier spécifié.
`--user=`*`utilisateur`* ou `-u` *`utilisateur`*	Désigne l'utilisateur devant se connecter.
`--verbose` ou `-v`	Mode verbeux pour afficher davantage de messages du serveur.
`--version` ou `-V`	Affiche la version du serveur et rend la main.
`--vertical` ou `-E`	Affiche les résultats des extractions verticalement (non plus en lignes horizontales).
`--xml` ou `-X`	Formate le résultat des extractions en XML. Les noms de balises générées sont `<resultset>` pour la table résultat, `<row>` pour chaque ligne et `<field>` pour les colonnes.

Ces options peuvent se combiner en les séparant simplement par un espace (exemple : `mysql --tee=D:\\dev\\sortiemysql.txt --database=bdsoutou` va se connecter anonymement à la base `bdsoutou` en inscrivant le contenu de la trace de la session dans le fichier `sortiemysql.txt` situé dans le répertoire `D:\dev`).

Batch

Pour lancer plusieurs commandes regroupées dans un fichier ayant pour extension .`sql`, il faut préciser le chemin du fichier et celui qui contiendra les éventuels résultats. Ainsi, l'instruction suivante exécute dans la base `bdsoutou`, sous l'autorité de l'utilisateur `soutou`, les commandes contenues dans le fichier `Testbatch.sql` situé dans le répertoire `D:\dev` (notez l'utilisation du double *back-slash* pour désigner une arborescence Windows). Le résultat sera consigné dans le fichier `sortie.txt` du même répertoire.

```
mysql --user=soutou --password=iut bdsoutou
                <D:\\dev\\Testbatch.sql >D:\\dev\\sortie.txt
```

Ici, la saisie du mot de passe n'est pas possible et il convient de le passer en clair dans la commande.

Votre prompt, et vite !

L'exécution de l'instruction `mysql --prompt="(\u@\h) [\d]> " --user=root –p` dans une fenêtre de commande *shell* ou DOS connectera l'utilisateur `root` en lui demandant son mot de passe. L'invite de commande à l'affichage sera de la forme suivante : `(root@localhost) [bdsoutou]>` une fois que `root` aura sélectionné la base `bdsoutou` (par la commande `use nombase;`).

Le tableau suivant résume les principaux paramètres pour afficher les invites de commande (relatives à l'option `prompt`).

Tableau I-5 Principales options de la commande `mysql`

Option	Commentaire
\v	Version du serveur
\d	Base de données en cours d'utilisation
\h	Nom du serveur
\u	Nom d'utilisateur
\U	Nom d'utilisateur long (au format *nom@serveur*)
_	Un espace
\R	Heure (0 à 23)
\m	Minutes
\s	Secondes
\Y	Année sur quatre chiffres
\D	Date en cours
\c	Compteur d'instructions

Pour personnaliser l'invite de commandes, deux possibilités s'offrent à vous. La première consiste à créer la variable d'environnement de nom `MYSQL_PS1` et de valeur `(\u@\h) [\d] mysql>`. La seconde solution nécessite d'ajouter une entrée au fichier de configuration `my.ini`, les lignes suivantes sous la section `[mysql]` :

```
#mon prompt
prompt=(\\u@\\h) [\\d]\\_mysql>\\_
```

Par ailleurs, si `my.ini` se trouve dans `C:\ProgramData\MySQL`, vous aurez peut-être besoin de dupliquer ce fichier dans le répertoire `C:\Programmes\MySQL\MySQL Server x.x` afin de bénéficier de votre prompt personnalisé.

Une fois le serveur redémarré, en considérant que votre utilisateur se nomme `util` et la base `bdutil`, le prompt précédant chacune de vos commandes SQL devrait être le suivant : `(util@localhost) [bdutil] mysql>`.

Commandes de base

Une fois connecté, vous pouvez utiliser des commandes ou effectuer des copier-coller d'un éditeur de texte dans l'interface `mysql` (ce moyen de faire correspond plus à un environnement de test qui conviendra à l'apprentissage). Le tableau suivant résume les principales instructions pour manipuler le *buffer* d'entrée de l'interface.

Tableau I-6 Commandes de base du buffer d'entrée

Commande	Commentaire
?	Affichage des commandes disponibles.
`delimiter` *`chaîne`*	Modifie le délimiteur (par défaut `;`).
`use` *`nomBase`*	Rend une base de données courante.
`prompt` *`chaîne`*	Modifie l'invite de commande avec les paramètres vus précédemment.
`quit` ou `exit`	Quitte l'interface.
`source` *`cheminNomFichier.sql`*	Charge et exécute dans le buffer le contenu du fichier *`cheminNomFichier.sql`* (par exemple, `source D:\\dev\\Testbatch.sql` exécutera le script `Testbatch.sql` situé dans `D:\dev`).
`tee` *`nomFichierSortie`*	Création du fichier `nomFichierSortie` dans le répertoire `C:\Program Files\MySQL\MySQL Server n.n\bin` qui contiendra la trace de la session.

La commande `source` est très utile afin d'éviter les copier-coller de trop nombreuses instructions.

Partie I

SQL de base

Chapitre 1

Définition des données

Ce chapitre décrit les instructions SQL qui constituent l'aspect LDD (langage de définition des données). À cet effet, nous verrons notamment comment déclarer une table avec ses éventuels index et contraintes.

Tables relationnelles

Une table est créée en SQL par l'instruction CREATE TABLE, modifiée au niveau de sa structure par l'instruction ALTER TABLE et supprimée par la commande DROP TABLE.

Création d'une table [CREATE TABLE]

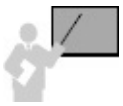
Pour pouvoir créer une table dans votre base, il faut que vous ayez reçu le privilège CREATE. Le mécanisme des privilèges est décrit au chapitre 5.

La syntaxe SQL simplifiée est la suivante :

```
CREATE [TEMPORARY] TABLE [IF NOT EXISTS] [nomBase.]nomTable
    ( colonne1 type1
          [NOT NULL | NULL] [DEFAULT valeur1] [COMMENT 'chaine1']
     [, colonne2 type2
          [NOT NULL | NULL] [DEFAULT valeur2] [COMMENT 'chaine2'] ]
     [CONSTRAINT nomContrainte1 typeContrainte1] ...)
     [ENGINE= InnoDB | MyISAM | ...];
```

- TEMPORARY : pour créer une table qui n'existera que durant la session courante (la table sera supprimée à la déconnexion). Deux connexions peuvent ainsi créer deux tables temporaires de même nom sans risquer de conflit. Il faut posséder le privilège CREATE TEMPORARY TABLES.
- IF NOT EXISTS : permet d'éviter qu'une erreur se produise si la table existe déjà (si c'est le cas, elle n'est aucunement affectée par la tentative de création).

- *nomBase* : (jusqu'à 64 caractères permis dans un nom de répertoire ou de fichier sauf « / », « \ » et « . ») s'il est omis, il sera assimilé à la base connectée. S'il est précisé, il désigne soit la base connectée soit une autre base (dans ce cas, il faut que l'utilisateur courant ait le droit de créer une table dans l'autre base). Nous aborderons ces points dans le chapitre Contrôle des données et nous considérerons jusque-là que nous travaillons dans la base courante (ce sera votre configuration la plupart du temps).
- *nomTable* : mêmes limitations que pour le nom de la base.
- *colonnei typei* : nom d'une colonne (mêmes caractéristiques que pour les noms des tables) et son type (INTEGER, CHAR, DATE…). Nous verrons quels types sont disponibles sous MySQL. La directive DEFAULT fixe une valeur par défaut. La directive NOT NULL oblige la colonne à contenir une donnée pour chaque ligne (information obligatoire). À l'inverse, NULL permet à la colonne de ne pas être renseignée pour toutes les lignes.



NULL n'est pas une valeur mais un concept (on parle de « marqueur » et non de valeur) qui permet de signifier « non disponible », « non affecté », « inconnu » ou « inapplicable ». NULL n'est pas la chaîne vide pour un caractère ou zéro pour un nombre. Vous devrez considérer avec attention toutes les données facultatives car, dans une comparaison SQL, deux NULL ne sont pas considérés comme identiques. Par ailleurs, pour d'autres opérateurs (regroupements ou ensemblistes), deux NULL sont, cette fois, considérés comme identiques.

- COMMENT : (jusqu'à 60 caractères) permet de commenter une colonne. Ce texte sera ensuite automatiquement affiché à l'aide des commandes SHOW CREATE TABLE et SHOW FULL COLUMNS (voir le chapitre 5).
- *nomContraintei typeContraintei* : nom de la contrainte et son type (clé primaire, clé étrangère, etc.). Nous allons détailler dans le paragraphe suivant les différentes contraintes possibles.
- ENGINE : définit le type de table (par défaut InnoDB, bien adapté à la programmation de transactions et adopté dans cet ouvrage). Le moteur MyISAM est très utilisé du fait de sa robustesse, mais il ne supporte ni les transactions ni l'intégrité référentielle. D'autres types existent, citons MEMORY pour les tables temporaires, ARCHIVE, etc.

- « ; » : symbole par défaut qui termine une instruction MySQL en mode ligne de commande (en l'absence d'un autre délimiteur).

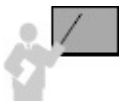
Délimiteurs

En mode ligne de commande, il est possible (par la directive `delimiter`) de choisir le symbole qui terminera chaque instruction. Dans l'exemple suivant on choisit le dollar ; au cours de l'ouvrage nous resterons avec le symbole par défaut de MySQL à savoir « ; ».

```
delimiter $
CREATE TABLE Test (t CHAR(8))$
```

Sensibilité à la casse

Alors que MySQL est sensible par défaut à la casse (au niveau des noms de base et de table) dans la plupart des distributions Unix, il ne l'est pas pour Windows ! En revanche, concernant les noms de colonnes, index, alias de colonnes, déclencheurs et procédures cataloguées, MySQL n'est pas sensible à la casse tous systèmes confondus. En fait, tous ces noms sont stockés en minuscules dans le dictionnaire de données.



La variable `lower_case_table_names` permet de forcer la sensibilité à la casse pour les noms des tables et des bases de données (si elle vaut 0, la sensibilité à la casse est active et les noms sont stockés en minuscules ; 1, pas de sensibilité à la casse et les noms sont stockés en minuscules ; 2, pas de sensibilité à la casse et les noms sont stockés en respectant la casse).

Je vous invite à positionner cette variable à 0 de manière à homogénéiser le codage et à contrôler un peu plus l'écriture de vos instructions SQL. De plus, c'est l'option par défaut sur Linux. Dans le fichier `my.ini`, sous la section serveur identifiée par [`mysqld`], ajouter la ligne et le commentaire suivants :

```
# Rend sensible à la CASSE les noms de tables et de database
lower_case_table_names=0
```

Refusez ce genre d'écriture (rendue impossible d'ailleurs si la variable `lower_case_table_names` est positionnée à 0).

```
mysql> SELECT * FROM Avion WHERE AVION.capacite > 150;
```

Par ailleurs, la casse devrait être respectée dans les expressions de comparaison entre colonnes et valeurs, que ce soit dans une instruction SQL ou un test dans un programme.

En fonction du jeu de caractères et de la collation que vous aurez choisie (ou subie), la condition `nomComp='Air France'` peut avoir la même signification que `nomComp='AIR France'`.

Si vous désirez vérifier la casse au sein même des données, il faudra choisir une collation qui le permet ou bien utiliser la fonction `BINARY()` pour convertir en bits une expression. Ainsi, `BINARY('AIR France')` sera toujours différent de `BINARY('Air France')`.

Commentaires

Dans toute instruction SQL (déclaration, manipulation, interrogation et contrôle des données), il est possible d'inclure des retours chariot, des tabulations, espaces et commentaires (sur une ligne précédée de deux tirets `- -`, en fin de ligne à l'aide du dièse #, au sein d'une ligne ou sur plusieurs lignes entre `/*` et `*/`). Les scripts suivants décrivent la déclaration d'une même table en utilisant différentes conventions :

Tableau 1-1 Différentes écritures SQL

Sans commentaire	Avec commentaires
`CREATE TABLE Test(colonne DECIMAL(38,8));`	`CREATE TABLE -- nom de la table Test( #début de la description COLONNE DECIMAL(38,8) )`
`CREATE TABLE`	

```
Test
(colonne
DECIMAL(38,8)
);
```

```
-- fin, ne pas oublier le point-virgule.
;
CREATE TABLE Test (
    /* une plus grande description
       des colonnes */
     COLONNE /* type : */ DECIMAL(38,8));
```

Cet ouvrage utilise les conventions suivantes :

- Tous les mots-clés de SQL sont notés en `majuscules`.
- Les noms de tables sont notés en `Minuscules` (excepté la première lettre, ces noms seront quand même stockés dans le système en minuscules).
- Les noms de colonnes et de contraintes en `minuscules`.

Premier exemple

Le tableau ci-après décrit l'instruction SQL qui permet de créer la table `Compagnie` illustrée par la figure suivante, dans la base *bdutil* (l'absence du préfixe « `bdutil`. » conduirait au même résultat si *bdutil* était la base connectée lors de l'exécution du script).

Figure 1-1 *Table à créer*

Compagnie

comp	nrue	rue	ville	nomComp

Tableau 1-2 Création d'une table et de ses contraintes

Instruction SQL	Commentaires

`CREATE TABLE bdutil.Compagnie`
`(comp CHAR(4),`
`nrue INTEGER(3),`
`rue VARCHAR(20),`
`ville VARCHAR(15)`
	La table contient cinq colonnes (quatre chaînes de caractères et un numérique de trois chiffres). La colonne `ville` est commentée.
	La table inclut en plus deux contraintes :

```
    DEFAULT 'Paris'
    COMMENT 'Par defaut : Paris',
nomComp  VARCHAR(15) NOT NULL);
```

- `DEFAULT` qui fixe *Paris* comme valeur par défaut de la colonne `ville` ;
- `NOT NULL` qui impose une valeur non nulle dans la colonne `nomComp`.

Contraintes

Les contraintes ont pour but de programmer des règles de gestion au niveau des colonnes des tables. Elles peuvent alléger un développement côté client (si on déclare qu'une note doit être comprise entre 0 et 20, les programmes de saisie n'ont plus à tester les valeurs en entrée mais seulement le code retour après connexion à la base ; on déporte les contraintes côté serveur).

Les contraintes peuvent être déclarées de deux manières.

- En même temps que la colonne (valable pour les contraintes monocolonnes) ; ces contraintes sont dites « en ligne » (*inline constraints*). L'exemple précédent en déclare deux.
- Après que la colonne est déclarée ; ces contraintes ne sont pas limitées à une colonne et peuvent être personnalisées par un nom (*out-of-line constraints*).

Il est recommandé de déclarer les contraintes `NOT NULL` en ligne, les autres peuvent soit être déclarées en ligne, soit être nommées. Étudions à présent les types de contraintes nommées (*out-of-line*). Les quatre types de contraintes les plus utilisées sont les suivants :

```
CONSTRAINT nomContrainte
UNIQUE (colonne1 [,colonne2]...)
PRIMARY KEY (colonne1 [,colonne2]...)
FOREIGN KEY (colonne1 [,colonne2]...)
REFERENCES nomTablePere [(colonne1 [,colonne2]...)]
              [ON DELETE {RESTRICT | CASCADE | SET NULL | NO ACTION}]
              [ON UPDATE {RESTRICT | CASCADE | SET NULL | NO ACTION}]
CHECK (condition)
```

- La contrainte `UNIQUE` impose une valeur distincte au niveau de la table (les valeurs nulles font exception à moins que `NOT NULL` soit aussi appliquée sur les colonnes).
- La contrainte `PRIMARY KEY` déclare la clé primaire de la table. Un index est généré automatiquement sur la ou les colonnes concernées. Les colonnes

clés primaires ne peuvent être ni nulles ni identiques (en totalité si elles sont composées de plusieurs colonnes).

- La contrainte `FOREIGN KEY` déclare une clé étrangère entre une table enfant (*child*) et une table père (*parent*). Ces contraintes définissent l'intégrité référentielle que nous aborderons plus tard. Les directives `ON UPDATE` et `ON DELETE` disposent de quatre options que nous détaillerons avec les directives `MATCH` à la section « Intégrité référentielle » du chapitre 2).

La contrainte `CHECK` impose un domaine de valeurs à une colonne ou une condition (exemples : `CHECK (note BETWEEN 0 AND 20)`, `CHECK (grade='Copilote' OR grade='Commandant')`). Il est permis de déclarer des contraintes `CHECK` lors de la création d'une table mais le mécanisme de vérification des valeurs n'est toujours pas implémenté.

Dans le manuel de référence (http://dev.mysql.com/doc/refman/x.y/en/alter-table.html), il est dit que la clause `CHECK` est comprise par MySQL mais non interprétée par tous les moteurs de stockage pour des raisons de compatibilité de portage de code vers un autre SGBD.

Il est recommandé de ne pas définir de contraintes sans les nommer (bien que cela soit possible), car il sera difficile de les faire évoluer (désactivation, réactivation, suppression), et la lisibilité des programmes en sera affectée.

Nous verrons au chapitre 3 comment ajouter, supprimer, désactiver et réactiver des contraintes (options de la commande `ALTER TABLE`).

Conventions recommandées

Adoptez les conventions d'écriture suivantes pour vos contraintes :

- Préfixez par `pk_` le nom d'une contrainte clé primaire, `fk_` une clé étrangère, `ck_` une vérification, `un_` une unicité.
- Pour une contrainte clé primaire, suffixez du nom de la table la contrainte (exemple `pk_Avion`).
- Pour une contrainte clé étrangère, renseignez (ou abrégez) les noms de la table source, de la clé, et de la table cible (exemple `fk_Pil_compa_Comp`).

En respectant nos conventions, déclarons les tables de l'exemple suivant (`Compagnie` avec sa clé primaire et `Avion` avec sa clé primaire et sa clé étrangère). Du fait de l'existence de la clé étrangère, la table `Compagnie` est dite « parent » (ou « père ») de la table `Avion` « enfant » (ou « fils »). Cela résulte de la traduction d'une association *un-à-plusieurs* entre les deux tables (*Modélisation de bases de données*, Eyrolles 2015). Nous reviendrons sur ces principes à la section « Intégrité référentielle » du prochain chapitre.

***Figure 1-2** Deux tables reliées à créer*

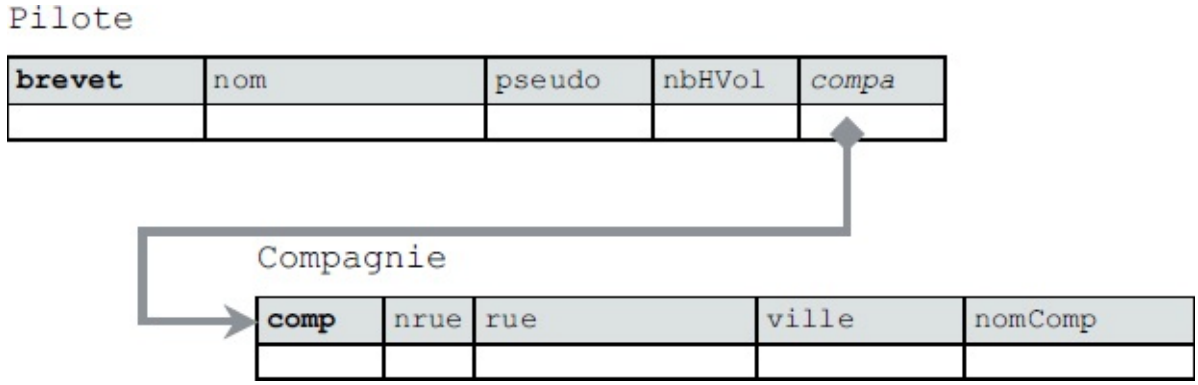


Tableau 1-3 Contraintes en ligne et nommées

Tables	Contraintes

`CREATE TABLE bdutil.Compagnie`
`(comp CHAR(4), nrue INTEGER(3),`
`rue VARCHAR(20), ville VARCHAR(15)`
`DEFAULT 'Paris' COMMENT 'Par defaut : Paris',`
`nomComp VARCHAR(15) NOT NULL,`
`CONSTRAINT pk_Compagnie PRIMARY KEY(comp));`
	Deux contraintes en ligne et une contrainte nommée de clé primaire.

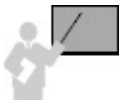
	Une contrainte en ligne et quatre

```
CREATE TABLE bdutil.Pilote
(brevet VARCHAR(6),   nom    VARCHAR(30) NOT NULL,
 pseudo VARCHAR(8),   nbHVol DECIMAL(7,2),
 compa CHAR(4),
 CONSTRAINT pk_Pilote PRIMARY KEY(brevet),
 CONSTRAINT ck_nbHVol CHECK (nbHVol BETWEEN 0 AND 20000),
 CONSTRAINT un_nom UNIQUE (nom),
 CONSTRAINT fk_Pil_compa_Comp FOREIGN KEY (compa)
           REFERENCES bdutil.Compagnie(comp));
```

contraintes nommées :

- Clé primaire
- NOT NULL
- CHECK (nombre d'heures de vol compris entre 0 et 20 000)
- UNIQUE (homonymes interdits)
- Clé étrangère

Remarques



- L'ordre n'est pas important dans la déclaration des contraintes nommées.
- PRIMARY KEY équivaut à : UNIQUE + NOT NULL + index.
- L'ordre de création des tables est important quand on définit les contraintes en même temps que les tables (on peut différer la création ou l'activation des contraintes, voir le chapitre 3). Il faut créer d'abord les tables « pères » puis les tables « fils ». Le script de destruction des tables suit le raisonnement inverse.

Types des colonnes

Pour décrire les colonnes d'une table, MySQL fournit les types prédéfinis suivants (*built-in datatypes*) :

- caractères (CHAR, VARCHAR, TINYTEXT, TEXT, MEDIUMTEXT, LONGTEXT) ;
- valeurs numériques (TINYINT, SMALLINT, MEDIUMINT, INT, INTEGER, BIGINT, FLOAT, DOUBLE, REAL, DECIMAL, NUMERIC, et BIT) ;
- date/heure (DATE, DATETIME, TIME, YEAR, TIMESTAMP) ;
- données binaires (BLOB, TINYBLOB, MEDIUMBLOB, LONGBLOB) ;

- énumérations (ENUM, SET).

Détaillons à présent ces types. Nous verrons comment utiliser les plus courants au chapitre 2 et les autres au fil de l'ouvrage.

Caractères

Le type CHAR permet de stocker des chaînes de caractères de taille fixe. Les valeurs sont stockées en ajoutant, s'il le faut, des espaces (*trailing spaces*) à concurrence de la taille définie. Ces espaces ne seront pas considérés après extraction à partir de la table.

Le type VARCHAR permet de stocker des chaînes de caractères de taille variable. Les valeurs sont stockées sans l'ajout d'espaces à concurrence de la taille définie. Depuis la version 5.0.3 de MySQL, les éventuels espaces de fin de chaîne seront stockés et extraits en conformité avec la norme SQL. Des caractères Unicode (méthode de codage universelle qui fournit une valeur de code unique pour chaque caractère quels que soient la plate-forme, le programme ou la langue) peuvent aussi être stockés.

Les types BINARY et VARBINARY sont similaires à CHAR et VARCHAR, excepté par le fait qu'ils contiennent des chaînes d'octets sans tenir compte d'un jeu de caractères en particulier.

Les quatre types permettant aussi de stocker du texte sont TINYTEXT, TEXT, MEDIUMTEXT, et LONGTEXT. Ces types sont associés à un jeu de caractères. Il n'y a pas de mécanisme de suppression d'espaces de fin ni de possibilité d'y associer une contrainte DEFAULT.

Tableau 1-4 Types de données caractères

Type	Description	Commentaire pour une colonne
`CHAR(n) [BINARY | ASCII | UNICODE]`	Chaîne fixe de *n* octets ou caractères.	Taille fixe (maximum de 255 caractères).
`VARCHAR(n) [BINARY]`	Chaîne variable de *n* caractères ou octets.	Taille variable (maximum de 65 535 caractères).

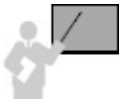
BINARY(*n*)	Chaîne fixe de *n* octets.	Taille fixe (maximum de 255 octets).
VARBINARY(*n*)	Chaîne variable de *n* octets.	Taille variable (maximum de 255 octets).
TINYTEXT(*n*)	Flot de *n* octets.	Taille fixe (maximum de 255 octets).
TEXT(*n*)	Flot de *n* octets.	Taille fixe (maximum de 65 535 octets).
MEDIUMTEXT(*n*)	Flot de *n* octets.	Taille fixe (maximum de 16 mégaoctets).
LONGTEXT(*n*)	Flot de *n* octets.	Taille fixe (maximum de 4,29 gigaoctets).

Valeurs numériques

De nombreux types sont proposés par MySQL pour définir des valeurs exactes (entiers ou décimaux positifs ou négatifs : INTEGER et SMALLINT), et des valeurs à virgule fixe ou flottante (FLOAT, DOUBLE et DECIMAL). En plus des spécifications de la norme SQL, MySQL propose les types d'entiers restreints (TINYINT, MEDIUMINT et BIGINT). Le tableau suivant décrit ces types :

- *n* indique le nombre de positions de la valeur à l'affichage (le maximum est de 255). Ainsi, il est possible de déclarer une colonne TINYINT(4) sachant que seules 3 positions sont nécessaires en fonction du domaine de valeurs permises.
- La directive UNSIGNED permet de considérer seulement des valeurs positives.
- La directive ZEROFILL complète par des zéros à gauche une valeur (par exemple : soit un INTEGER(5) contenant 4, si ZEROFILL est appliqué, la valeur extraite sera « 00004 »). En déclarant une colonne ZEROFILL, MySQL l'affecte automatiquement à UNSIGNED aussi.

Synonymes et alias



- INT est synonyme de INTEGER.

- `DOUBLE PRECISION` et `REAL` sont synonymes de `DOUBLE`.
- `DEC NUMERIC` et `FIXED` sont synonymes de `DECIMAL`.
- `SERIAL` est un alias pour `BIGINT UNSIGNED NOT NULL AUTO_INCREMENT UNIQUE`.
- Dans toute instruction SQL, écrivez la virgule avec un point (7/2 retourne 3.5).

Tableau 1-5 Types de données numériques

Type	Description
`BIT[(n)]`	Ensemble de *n* bits. Taille de 1 à 64 (par défaut 1).
`TINYINT[(n)] [UNSIGNED] [ZEROFILL]`	Entier (sur un octet) de -128 à 127 signé, 0 à 255 non signé.
`BOOL` et `BOOLEAN`	Synonymes de `TINYINT(1)`, la valeur zéro est considérée comme fausse. Le non-zéro est considéré comme vrai. Dans les prochaines versions, le type *boolean*, comme le préconise la norme SQL, sera réellement pris en charge.
`SMALLINT[(n)] [UNSIGNED] [ZEROFILL]`	Entier (sur 2 octets) de – 32 768 à 32 767 signé, 0 à 65 535 non signé.
`MEDIUMINT[(n)] [UNSIGNED] [ZEROFILL]`	Entier (sur 3 octets) de – 8 388 608 à 8 388 607 signé, 0 à 16 777 215 non signé.
`INTEGER[(n)] [UNSIGNED] [ZEROFILL]`	Entier (sur 4 octets) de –2 147 483 648 à 2 147 483 647 signé, 0 à 4 294 967 295 non signé.
`BIGINT[(n)] [UNSIGNED] [ZEROFILL]`	Entier (sur 8 octets) de – 9 223 372 036 854 775 808 à 9 223 372 036 854 775 807 signé, 0 à 18 446 744 073 709 551 615 non signé.
`FLOAT[(n[,p])] [UNSIGNED] [ZEROFILL]`	Flottant (de 4 à 8 octets) *p* désigne la précision simple (jusqu'à 7décimales) de $-3.4\ 10^{+38}$ à $-1.1\ 10^{-38}$, 0, signé, et de $1.1\ 10^{-38}$ à $3.4\ 10^{+38}$ non signé.
`DOUBLE[(n[,p])] [UNSIGNED]`	Flottant (sur 8 octets) *p* désigne la précision double (jusqu'à 15 décimales) de $-1.7\ 10^{+308}$

`[ZEROFILL]`	à -2.2 10^{-308}, 0, signé, et de 2.2 10^{-308} à 1.7 10^{+308} non signé.
`DECIMAL[(n[,p])] [UNSIGNED] [ZEROFILL]`	Décimal (sur 4 octets) à virgule fixe, *p* désigne la précision (nombre de chiffres après la virgule, maximum 30). Par défaut *n* vaut 10, *p* vaut 0. Nombre maximal de chiffres pour un décimal : 65.

Dates et heures

Les types suivants permettent de stocker des moments ponctuels (dates, dates et heures, années, et heures). Les fonctions `NOW`() et `SYSDATE`() retournent la date et l'heure courantes. Dans une procédure ou un déclencheur `SYSDATE` est réévaluée en temps réel, alors que `NOW` désignera toujours l'instant de début de traitement.

Données binaires

Les types `BLOB` (*Binary Large OBject*) permettent de stocker des données non structurées comme le multimédia (images, sons, vidéo, etc.). Les quatre types de colonnes `BLOB` sont `TINYBLOB`, `BLOB`, `MEDIUMBLOB` et `LONGBLOB`. Ces types sont traités comme des flots d'octets sans jeu de caractère associé.

Tableau 1-6 Types de données dates et heures

Type	Description	Commentaire pour une colonne
`DATE`	Dates du 1er janvier de l'an 1000 au 31 décembre 9999 après J.-C.	Sur 3 octets. L'affichage est au format `'YYYY-MM-DD'`.
`DATETIME`	Dates et heures (de 0 h de la première date à 23 h 59 minutes 59 secondes de la dernière date).	Sur 8 octets. L'affichage est au format `'YYYY-MM-DD HH:MM:SS'`.
`YEAR[(2|4)]`	Sur 4 positions : de 1901 à 2155 (incluant 0000). Sur 2 positions (autorisé jusqu'en version 5.5) : de 70 à 69 désignant 1970 à 2069.	Sur 1 octet ; l'année est considérée sur 2 ou 4 positions (4 par défaut). Le format d'affichage est `'YYYY'`.

`TIME`	Heures de -838 h 59 minutes 59 secondes à 838 h 59 minutes 59 secondes.	L'heure au format `'HHH:MM:SS'` sur 3 octets.
`TIMESTAMP`	Instants du 1er Janvier 1970 0 h 0 minute 0 seconde à l'année 2037.	Estampille sur 4 octets (au format `'YYYY-MM-DD HH:MM:SS'`) ; mise à jour à chaque modification sur la table.

Tableau 1-7 Types de données binaires

Type	Description	Commentaire pour une colonne
`TINYBLOB(`*n*`)`	Flot de *n* octets.	Taille fixe (maximum de 255 octets).
`BLOB(`*n*`)`		Taille fixe (maximum de 65 535 octets).
`MEDIUMBLOB(`*n*`)`		Taille fixe (maximum de 16 mégaoctets).
`LONGBLOB(`*n*`)`		Taille fixe (maximum de 4,29 gigaoctets).

Énumération

Deux types de collections sont proposés par MySQL.

- Le type `ENUM` définit une liste de valeurs permises (chaînes de caractères).
- Le type `SET` permettra de comparer une liste à une combinaison de valeurs permises à partir d'un ensemble de référence (chaînes de caractères).

Tableau 1-8 Types de données énumération

Type	Description
`ENUM('valeur1','valeur2',...)`	Liste de 65 535 valeurs au maximum.
`SET('valeur1','valeur2',...)`	Ensemble de référence (maximum de 64 valeurs).

Structure d’une table [DESCRIBE]

DESCRIBE (écriture autorisée DESC) est une commande qui vient de SQL*Plus d’Oracle et qui a été reprise par MySQL. Elle permet d’extraire la structure brute d’une table ou d’une vue.

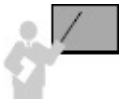
```
DESCRIBE [nomBase.] nomTableouVue [colonne];
```

Si la base n’est pas indiquée, il s’agit de celle en cours d’utilisation. Retrouvons la structure des tables Compagnie et Pilote précédemment créées. Le type de chaque colonne apparaît :

Tableau 1-9 Structure brute des tables

Résultat	Commentaires
mysql> **DESCRIBE** bdutil.Pilote; +--------+-------------+------+-----+---------+-------+ | Field | Type | Null | Key | Default | Extra | +--------+-------------+------+-----+---------+-------+ | brevet | varchar(6) | NO | PRI | NULL | | | nom | varchar(30) | NO | UNI | NULL | | | pseudo | varchar(8) | YES | | NULL | | | nbHVol | decimal(7,2) | YES | | NULL | | | compa | char(4) | YES | MUL | NULL | | +--------+-------------+------+-----+---------+-------+	Les clés primaires sont NOT NULL (désignées par PRI dans la colonne Key). Les unicités sont désignées par UNI dans la colonne Key. Les occurrences multiples possibles sont désignées par MUL dans la colonne Key. Les contraintes NOT NULL nommées (définies via les contraintes CHECK) n’apparaissent pas.
mysql> **DESCRIBE** bdutil Compagnie; +---------+-------------+------+-----+---------+-------+ | Field | Type | Null | Key | Default | Extra | +---------+-------------+------+-----+---------+-------+ | comp | char(4) | NO | PRI | | | | nrue | int(3) | YES | | NULL | | | rue | varchar(20) | YES | | NULL | | | ville | varchar(15) | YES | | Paris | | | nomComp | varchar(15) | NO | | | | +---------+-------------+------+-----+---------+-------+	

La colonne `Extra` indique notamment les séquences (`AUTO_INCREMENT`).



La commande `SHOW CREATE TABLE [nom_base.]nom_table;` (voir le chapitre 5) restitue l'instruction complète qui a permis la création de la table en question.

La commande `SHOW FULL COLUMNS FROM [nom_base.]nom_table;` est plus complète que l'instruction `DESCRIBE`.

Restrictions

Les contraintes `CHECK` définies ne sont pas encore opérationnelles.

Les colonnes de type `SET` sont évaluées par des chaînes de caractères séparés par des « , » ('Airbus, Boeing'). En conséquence aucune valeur d'un SET ne doit contenir le symbole « , ».

Les noms des objets (base, tables, colonnes, contraintes, vues, etc.) ne doivent pas emprunter des mots-clés de MySQL : `TABLE`, `SELECT`, `INSERT`, `IF`… Si vous êtes « franco-français » cela ne vous gênera pas.

Les collations et jeux de caractères

Une collation est liée à un jeu de caractères et permet de classer les caractères dans le jeu (lors d'un tri ou d'une comparaison). Par exemple, il sera possible de différencier « à » de « a » (sensibilité diacritique). Chaque jeu de caractères possède plusieurs collations, dont une par défaut. Toutes les collations ont un nom qui commence par le jeu de caractères auquel elles sont liées et se termine par `_bin` (*binary*), `_cs` (*case sensitive*) ou `_ci` (*case insensitive*). Concernant les

collations binaires, les caractères sont ordonnés selon leurs numéros de code (d'abord les majuscules, puis les minuscules, puis les lettres accentuées). Les collations non binaires permettent de trier selon le langage, en choisissant la sensibilité à la casse.

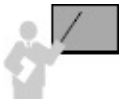
Le tableau suivant résume les caractéristiques de deux jeux de caractères : le latin qui concerne l'Europe de l'ouest et l'UTF8 qui est conforme au standard unicode.

Tableau 1-10 Caractéristiques de deux jeux de caractères

Jeu de caractères	Collation	Sensible à la casse	Sensible aux accents
latin1	latin1_bin	oui	oui
	latin1_general_cs		
	latin1_general_ci	non	
utf8	utf8_bin	oui	oui
	utf8_general_ci	non	non
	utf8_general_bin		

Dans l'exemple suivant, le jeu de caractères de la table est possitionné à `utf8`, mais deux colonnes sont fixées à `latin1` en assurant une sensibilité à la casse pour le nom de la compagnie.

```
CREATE TABLE bdutil.Compagnie (comp CHAR(4), nrue INTEGER(3),
                               rue VARCHAR(20),
  ville   VARCHAR(15) CHARACTER SET latin1 COLLATE latin1_general_ci,
  nomComp VARCHAR(15) CHARACTER SET latin1 COLLATE latin1_general_cs
 ) DEFAULT CHARACTER SET utf8 COLLATE utf8_bin;
```



Les données utilisent le jeu de caractères et la collation de leur colonne. Si la collation n'a pas été spécifiée pour la colonne, elle adopte celle de la table. Si la collation de la table n'a pas été spécifiée, elle adopte la collation de la base. Si la collation de la base n'a pas été spécifiée non plus, le jeu de caractères et la collation par défaut sont ceux du serveur.

La commande `SHOW CHARACTER SET` vous donnera la liste des jeux de caractères à votre disposition et `SHOW VARIABLES LIKE 'collation%'` indiquera les collations par défaut au niveau de la connexion, de la base et du serveur.

Index

Comme l'index de cet ouvrage vous aide à atteindre les pages concernées par un mot recherché, un index MySQL permet d'accélérer l'accès aux données d'une table. Le but principal d'un index est d'éviter de parcourir une table séquentiellement du premier enregistrement jusqu'à celui visé (problème rencontré si c'est le Français nommé « Zidane » qu'on recherche dans une table non indéxée de plus de soixante-six millions d'enregistrements…). Le principe d'un index est l'association de l'adresse de chaque enregistrement avec la valeur des colonnes indéxées.

Sans index, tous les enregistrements de tous les blocs qui constituent la table seront systématiquement parcourus, et ce quelque soit la requête écrite (même celle qui doit retrouver une seule ligne de la table). Avec un index, quelques parcours de blocs peuvent suffire et ce nombre d'accès sera décorrélé de la montée en charge des enregistrements. En effet, si une table grossit d'un facteur 10 000, le temps de traitement sera proportionnel sans index, alors qu'il n'augmentera éventuellement que d'un point de vue logarithmique (ici, schématiquement 4 blocs de plus parcourus au lieu de 10 000 fois plus d'accès). Le parcours d'un index est dichotomique et le moyen le plus efficace de rechercher une valeur parmi un ensemble.

Arbres balancés

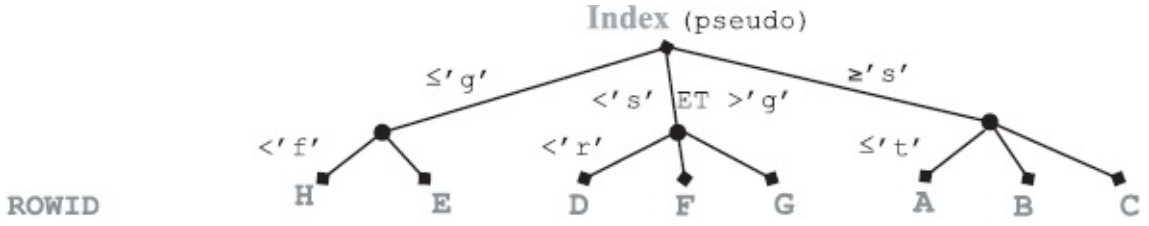
La figure suivante illustre un index unique sous la forme d'un arbre. Cet index est basé sur la colonne `pseudo` de la table `Pilote`. Dans cet exemple, trois accès à l'index seront nécessaires pour adresser directement un pilote via son pseudo, au lieu d'en analyser huit au plus. Plus il y aura de lignes dans la tables, plus l'index sera utile.

Un index est associé à une table et peut être défini sur une ou plusieurs colonnes (dites « indéxées »). Une table peut « héberger » plusieurs index. Ils sont mis à jour automatiquement après rafraîchissement de la table (ajouts et suppressions d'enregistrements ou modification des colonnes indéxées). Un index peut être déclaré unique si on sait que les valeurs des colonnes indéxées seront toujours uniques.

La plupart des index de MySQL (`PRIMARY KEY`, `UNIQUE`, `INDEX`, et `FULLTEXT`) sont stockés dans des arbres équilibrés (*balanced trees* : *B-trees*). D'autres types d'index

existent, citons ceux qui portent sur des colonnes SPATIAL (*reverse key : R-trees*), et ceux appliqués aux tables MEMORY (tables de hachage : *hash*).

***Figure 1-3** Index sur la colonne nom*



Pilote

ROWID	**brevet**	nom	pseudo	nbHVol	compa
A	PL-1	Ferrage	shark	4500	AF
B	PL-2	Sayag	vanier	9000	AF
C	PL-3	Giaconne	vip	1000	SING
D	PL-4	Aubrun	picsou	8500	ALIB
E	PL-5	Alquie	footy	5000	SING
F	PL-6	Benech	roma	2500	SING
G	PL-7	Roux	rox	6000	ALIB
H	PL-8	Mansalier	chips	4000	AF

La particularité des index *B-tree* est qu'ils conservent en permanence une arborescence symétrique (balancée). Toutes les feuilles sont à la même profondeur. Le temps de recherche est ainsi à peu près constant quel que soit l'enregistrement cherché. Le plus bas niveau de l'index (*leaf blocks*) contient les valeurs des colonnes indexées et le *rowid*. Toutes les feuilles de l'index sont chaînées entre elles. Pour les index non uniques (par exemple si on voulait définir un index sur la colonne compa de la table Pilote) le *rowid* est inclus dans la valeur de la colonne indexée. Ces index, premiers apparus, sont désormais très fiables et performants, ils ne se dégradent pas lors de la montée en charge de la table.

Création d'un index [CREATE INDEX]

Pour pouvoir créer un index dans une base, la table à indexer doit appartenir à la base en question. Si l'utilisateur a le privilège INDEX, il peut créer et supprimer des index dans sa base. Un index est créé par l'instruction CREATE INDEX et supprimé par DROP INDEX.

La syntaxe de création d'un index est la suivante :

```
CREATE [UNIQUE | FULLTEXT | SPATIAL] INDEX nomIndex
    [USING BTREE | HASH]
    ON [nomBase.]nomTable (colonne1 [(taille1)] [ASC | DESC],...) ;
```

- `UNIQUE` permet de créer un index qui n'accepte pas les doublons.
- `FULLTEXT` permet de bénéficier de fonctions de recherche dans des textes (flot de caractères).
- `SPATIAL` permet de profiter de fonctions pour les données géographiques.
- `ASC` et `DESC` précisent l'ordre (croissant ou décroissant).

Créons deux index sur la table `Pilote`.

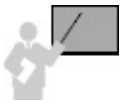
Tableau 1-11 Créations d'index

Instruction SQL	Commentaires

`USE bdutil;`
`CREATE UNIQUE INDEX idx_Pilote_nom3`
`  USING BTREE`
`  ON Pilote (nom(3) DESC);`
	Index *B-tree*, ordre décroissant sur les trois premiers caractères du nom des pilotes.

`CREATE INDEX idx_Pilote_compa`
`  USING BTREE`
`  ON Pilote (compa);`
	Index *B-tree*, ordre croissant sur la colonne clé étrangère `compa`.

Bilan



- Un index ralentit les rafraîchissements de la base (conséquence de la mise à jour de l'arbre ou des *bitmaps*). En revanche il accélère les accès.
- Il est conseillé de créer des index sur des colonnes (majoritairement des clés étrangères) utilisées dans les clauses de jointures (voir chapitre 4).
- Il est possible de créer des index pour toutes les colonnes d'une table (jusqu'à concurrence de 16).
- Les index sont pénalisants lorsqu'ils sont définis sur des colonnes très souvent modifiées ou si la table contient peu de lignes.

Destruction d'un schéma

Il vous sera utile d'écrire un script de destruction d'un schéma (j'entends « schéma » comme ensemble de tables, contraintes et index composant une base de données et non pas en tant qu'ensemble de tous les objets d'un utilisateur) pour pouvoir recréer une base propre. Bien entendu, si des données sont déjà présentes dans les tables, et que vous souhaitez les garder, il faudra utiliser une stratégie pour les réimporter dans les nouvelles tables. À ce niveau de l'ouvrage, vous n'en êtes pas là, et le script de destruction va vous permettre de corriger vos erreurs de syntaxe du script de création des tables.

Nous avons vu qu'il fallait créer d'abord les tables « pères » puis les tables « fils » (si des contraintes sont définies en même temps que les tables). L'ordre de destruction des tables, pour des raisons de cohérence, est inverse (il faut détruire les tables « fils » puis les tables « pères »). Dans notre exemple, il serait malvenu de supprimer la table `Compagnie` avant la table `Pilote`. En effet la clé étrangère `compa` n'aurait plus de sens.

Suppression d'une table [`DROP TABLE`]

Pour pouvoir supprimer une table dans une base, il faut posséder le privilège `DROP` sur cette base. L'instruction `DROP TABLE` entraîne la suppression des données, de la structure, de la description dans le dictionnaire des données, des index, des déclencheurs associés (*triggers*) et la récupération de la place dans l'espace de stockage.

```
DROP [TEMPORARY] TABLE [IF EXISTS]
    [nomBase.] nomTable1 [,[nomBase2.]nomTable2,...]
    [RESTRICT | CASCADE]
```

- `TEMPORARY` : pour supprimer des tables temporaires. Les transactions en cours ne sont pas affectées. L'utilisation de `TEMPORARY` peut être un bon moyen de s'assurer qu'on ne détruit pas accidentellement une table non temporaire…
- `IF EXISTS` : permet d'éviter qu'une erreur se produise si la table n'existe pas.
- `RESTRICT` (par défaut) permet de vérifier qu'aucun autre élément n'utilise la table (autre table via une clé étrangère, vue, déclencheur, etc.).

- `CASCADE`, option qui n'est pas encore opérationnelle, doit répercuter la destruction à toutes les autres tables référencées par des clés étrangères.

Les éléments qui utilisaient la table (vues, synonymes, fonctions et procédures) ne sont pas supprimés mais sont temporairement inopérants. Attention, une suppression ne peut pas être par la suite annulée.

Ordre des suppressions

Il suffit de relire à l'envers le script de création de vos tables pour en déduire l'ordre de suppression à écrire dans le script de destruction de votre schéma.

Le tableau suivant présente deux écritures possibles pour détruire le schéma contenant les tables `Pilote` et `Compagnie` et deux index. La première écriture ne se soucie pas de l'ordre des tables (il faut alors supprimer les clés étrangères, les index, puis les tables dans un ordre cohérent). La seconde écriture suit l'ordre des tables enfants, puis parents.

Tableau 1-12 Scripts de destruction

Sans ordre des tables	En respectant l'ordre des tables

`DROP INDEX idx_Pilote_nom3 ON bdutil.Pilote;`
`ALTER TABLE bdutil.Pilote`
`  DROP FOREIGN KEY fk_Pil_compa_Comp;`
`DROP INDEX idx_Pilote_compa`
`  ON bdutil.Pilote;`
`DROP TABLE bdutil.Compagnie;`
`DROP TABLE bdutil.Pilote;`
	`DROP TABLE bdutil.Pilote;`
	`DROP TABLE bdutil.Compagnie;`

Exercices

L'objectif de ces exercices est de créer des tables, leur clé primaire et des contraintes de vérification (`NOT NULL` et `CHECK`).

Exercice 1.1

Présentation de la base de données

Une entreprise désire gérer son parc informatique à l'aide d'une base de

données. Le bâtiment est composé de trois étages. Chaque étage possède son réseau (ou segment distinct) éthernet. Ces réseaux traversent des salles équipées de postes de travail. Un poste de travail est une machine sur laquelle sont installés certains logiciels. Quatre catégories de postes de travail sont recensées (stations Unix, terminaux X, PC Windows et PC NT). La base de données devra aussi décrire les installations de logiciels.

Les noms et types des colonnes sont les suivants :

Tableau 1-13 Caractéristiques des colonnes

Colonne	Commentaire	Type
indIP	trois premiers groupes IP (exemple : 130.120.80)	VARCHAR(11)
nomSegment	nom du segment	VARCHAR(20)
etage	étage du segment	TINYINT(1)
nSalle	numéro de la salle	VARCHAR(7)
nomSalle	nom de la salle	VARCHAR(20)
nbPoste	nombre de postes de travail dans la salle	TINYINT(2)
nPoste	code du poste de travail	VARCHAR(7)
nomPoste	nom du poste de travail	VARCHAR(20)
ad	dernier groupe de chiffres IP (exemple : 11)	VARCHAR(3)
typePoste	type du poste (UNIX, TX, PCWS, PCNT)	VARCHAR(9)
dateIns	date d'installation du logiciel sur le poste	dateTIME
nLog	code du logiciel	VARCHAR(5)
nomLog	nom du logiciel	VARCHAR(20)
dateAch	date d'achat du logiciel	dateTIME
version	version du logiciel	VARCHAR(7)
typeLog	type du logiciel (UNIX, TX, PCWS, PCNT)	VARCHAR(9)
prix	prix du logiciel	DECIMAL(6,2)
numIns	numéro séquentiel des installations	INTEGER(5)
dateIns	date d'installation du logiciel	TIMESTAMP
delai	intervalle entre achat et installation	SMALLINT
typeLP	types des logiciels et des postes	VARCHAR(9)
nomType	noms des types (Terminaux X, PC Windows…)	VARCHAR(20)

Exercice 1.2

Création des tables

Écrire puis exécuter le script SQL (que vous appellerez `creParc.sql`) de création des tables avec leur clé primaire (en gras dans le schéma suivant) et les contraintes suivantes :

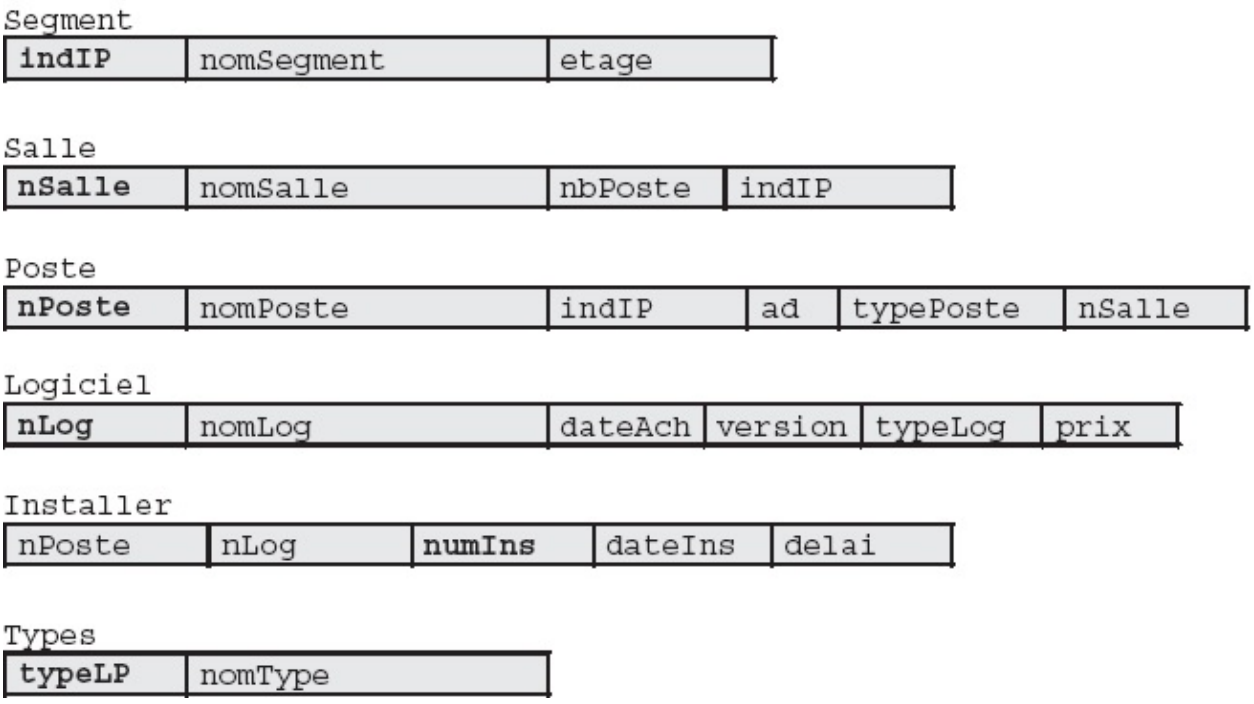
Les noms des segments, des salles et des postes sont non nuls.

Le domaine de valeurs de la colonne `ad` s'étend de 0 à 255.

La colonne `prix` est supérieure ou égale à 0.

La colonne `dateIns` est égale à la date du jour par défaut.

Figure 1-4 *Composition des tables*



Exercice 1.3

Structure des tables

Écrire puis exécuter le script SQL (que vous appellerez `descParc.sql`) qui affiche la description de toutes ces tables (en utilisant des commandes `DESCRIBE`). Comparer le résultat obtenu avec le schéma ci-dessus.

Exercice 1.4

Destruction des tables

Écrire puis exécuter le script SQL de destruction des tables (que vous appellerez `dropParc.sql`). Lancer ce script puis celui de la création des tables à nouveau.

Chapitre 2

Manipulation des données

Ce chapitre décrit l'aspect LMD (langage de manipulation des données) de MySQL. Nous verrons que SQL propose trois instructions pour manipuler des données :

- l'insertion d'enregistrements : `INSERT` ;
- la modification de données : `UPDATE` ;
- la suppression d'enregistrements : `DELETE` (et `TRUNCATE`).

Il existe d'autres possibilités pour insérer des données dans une base. Vous pouvez recourir à des commandes en ligne d'administration et de programmation (comme `LOAD DATA INFILE` que nous étudierons par la suite), ou à des outils d'importation et de migration, parmi lesquels nous pouvons citer *MySQL Workbench*, *phpMyAdmin*, *Navicat* et *EMS SQL Management Studio*.

Insertions d'enregistrements [INSERT]

Pour pouvoir insérer des enregistrements dans une table, il faut que vous ayez reçu le privilège `INSERT`. Il existe plusieurs possibilités d'insertion : l'insertion monoligne qui ajoute un enregistrement par instruction (que nous allons détailler maintenant) et l'insertion multiligne qui insère plusieurs enregistrements par une requête (que nous détaillerons au chapitre 4).

Syntaxe

La syntaxe simplifiée de l'instruction `INSERT` monoligne est la suivante :

```
INSERT [LOW_PRIORITY | DELAYED | HIGH_PRIORITY] [IGNORE]
    [INTO] [nomBase.] { nomTable | nomVue } [(nomColonne,...)]
    VALUES ({expression | DEFAULT},...),(...),...
    [ON DUPLICATE KEY UPDATE nomColonne = expression,...]
```

- DELAYED indique que l'insertion est différée (si la table est modifiée par ailleurs, le serveur attend qu'elle se libère pour y insérer périodiquement de nouveaux enregistrements si elle redevient active entre-temps).
- LOW_PRIORITY diffère l'exécution de la commande tant qu'il existe un client qui accède à la table. Cela ne concerne que les moteurs de stockage qui verrouillent au niveau table (MyISAM, MEMORY et MERGE).
- HIGH_PRIORITY annule l'option *low priority* du serveur.
- IGNORE indique que les éventuelles erreurs déclenchées suite à l'insertion seront considérées en tant que *warning*s.
- ON DUPLICATE KEY UPDATE permet de mettre à jour l'enregistrement présent dans la table, qui a déclenché l'erreur de doublon (dans le cas d'un index UNIQUE ou d'une clé primaire). Dans ce cas le nouvel enregistrement n'est pas inséré, seul l'ancien est mis à jour.

À l'aide d'exemples, nous allons détailler les possibilités de cette instruction en considérant la majeure partie des types de données proposés par MySQL.

Les doublons

Lorsqu'une insertion déclenche une valeur en doublon concernant une colonne unique (clé primaire ou contrainte UNIQUE), par défaut l'insertion est refusée (à juste titre). Ce comportement est décrit dans la section « Ne pas respecter les contraintes ».

En revanche, si l'option ON DUPLICATE KEY UPDATE est présente, le nouvel enregistrement n'est pas inséré, mais l'ancien peut être éventuellement modifié. Le script suivant présente cette option.

Tableau 2-1 Insertions avec un doublon

Instructions SQL	Commentaires
`CREATE TABLE SGBD` `  (a INTEGER(4) PRIMARY KEY, b CHAR(6));`	Création d'une table.

`INSERT INTO SGBD (a, b)`
`  VALUES (1995, 'Oracle');`
`INSERT INTO SGBD (a, b)`
`  VALUES (1995, 'MySQL')`
`  ON DUPLICATE KEY UPDATE a = a + 15;`
	Insertion de deux enregistrements dont le second en doublon.

`mysql> SELECT a,b FROM SGBD;` `+------+--------+` `| a	b |` `+------+--------+`	Le premier enregistrement est modifié ; le second enregistrement

```
| 2010 | Oracle |
+------+--------+
```
n’est pas en base.

Renseigner toutes les colonnes

Ajoutons trois lignes dans la table `Compagnie` en alimentant toutes les colonnes de la table par des valeurs. La deuxième insertion utilise le mot-clé `DEFAULT` pour affecter explicitement la valeur par défaut à la colonne `ville`. La troisième insertion attribue explicitement la valeur `NULL` à la colonne `nrue`.

Tableau 2-2 Insertions

Instruction SQL	Commentaires

`INSERT INTO Compagnie`
`  VALUES ('SING', 7, 'Camparols', 'Singapour',`
`  'Singapore AL');`
	Toutes les valeurs sont renseignées dans l’ordre de la structure de la table.

`INSERT INTO Compagnie`
`  VALUES ('AC', 10, 'Gambetta', **DEFAULT**,`
`  'Air France');`
	`DEFAULT` explicite.

`INSERT INTO Compagnie`
`  VALUES ('AN1', **NULL**, 'Hoche', 'Blagnac',`
`  'Air Nul1');`
	`NULL` explicite.

Renseigner certaines colonnes

Insérons deux lignes dans la table `Compagnie` en ne précisant pas toutes les colonnes. La première insertion affecte implicitement la valeur par défaut à la colonne `ville`. La deuxième donne implicitement la valeur `NULL` à la colonne `nrue`.

Tableau 2-3 Insertions de certaines colonnes

Instruction SQL	Commentaires
`INSERT INTO Compagnie(comp, nrue, rue, nomComp)` `  VALUES ('AF', 8, 'Champs Elysées', 'Castanet Air');`	`DEFAULT` implicite.
`INSERT INTO Compagnie(comp, rue, ville, nomComp)` `  VALUES ('AN2', 'Foch', 'Blagnac', 'Air Nul2');`	`NULL` sur `nrue` implicite.

La table `Compagnie` contient à présent les lignes suivantes :

Figure 2-1 *Table après les insertions*

Compagnie — Valeur NULL — Valeur par défaut

comp	nrue	rue	ville	nomComp
SING	7	Camparols	Singapour	Singapore AL
AF	10	Gambetta	**Paris**	Air France
AN1		Hoche	Blagnac	Air Nul1
AC	8	Champs Elysées	**Paris**	Castanet Air
AN2		Foch	Blagnac	Air Nul2

Renseignez vos colonnes !

Mis à part pour vos tests et démonstrations, renseignez systématiquement le nom de toutes les colonnes de la table dans laquelle vous insérez des enregistrements. Vous améliorerez grandement la qualité de votre code et réduirez la maintenance de vos programmes.

À ce titre, préférez toujours la première écriture à la seconde. Supposez que vous ajoutiez la colonne `mail` (non obligatoire) à la table. Le second type d'écriture ne sera plus exploitable…

Tableau 2-4 Comment insérer correctement

Insertion explicite	Insertion implicite
`INSERT INTO Pilote (brevet,pseudo,nom,nbHVol, compa) VALUES ('BE456','willy','Blanchet',8700,'SING');`	`INSERT INTO Pilote VALUES ('BE410','Castel', 'mulos', 3400, 'AF');`

Plusieurs enregistrements

L'instruction suivante ajoute trois nouvelles compagnies en une seule instruction `INSERT`. Dans ce cas, il n'est pas possible de renseigner les noms de colonnes.

```
INSERT INTO Compagnie VALUES
 ('LUFT',9,'Salas','Munich','Luftansa'),
 ('QUAN',1,'Kangouroo','Sydney','Quantas'),
 ('SNCM',3,'P. Paoli','Bastia','Corse Air');
```

Ne pas respecter des contraintes

Insérons des enregistrements dans la table `Pilote`, qui ne respectent pas des contraintes. Le tableau suivant décrit les messages renvoyés pour chaque erreur (le nom de la contrainte apparaît dans chaque message, les valeurs erronées sont notées en gras). La première erreur vient de la clé primaire, la seconde de l'unicité du nom. La troisième erreur signifie que la clé étrangère référence une clé primaire absente (ici une compagnie inexistante). Nous reviendrons sur ce dernier problème dans la section « Intégrité référentielle ». La dernière concerne la contrainte en ligne `NOT NULL`

Tableau 2-5 Insertions valides et invalides

Insertions vérifiant les contraintes	Insertions ne vérifiant pas les contraintes

`INSERT INTO Pilote (brevet,nom,pseudo,nbHVol, compa)`
`VALUES ('PL-1', 'Ente', 'loulou', 450, 'AF');`
	`mysql> INSERT INTO Pilote (brevet,nom, pseudo,nbHVol,compa)`
	`-> VALUES ('PL-1', 'Diffis', 'gege',`
	`900, 'AF');`
	`ERROR **1062** (23000): **Duplicate entry**`
	`'PL-1' for key 'PRIMARY'`

`INSERT INTO Pilote (brevet,nom,pseudo,nbHVol, compa)`
`VALUES ('PL-2', 'Ferrage', 'jpf', 900, 'AF');`
	`mysql> INSERT INTO Pilote (brevet,nom, pseudo,nbHVol,compa)`
	`-> VALUES ('PL-4', 'Ente', 'juju',`
	`1050, 'AF');`
	`ERROR **1062** (23000): **Duplicate entry**`
	`'Ente' for key 'un_nom'`

`INSERT INTO Pilote (brevet,nom,pseudo,nbHVol, compa)`
`VALUES ('PL-3', 'Soutou','polo', 1000, 'SING');`
	`mysql> INSERT INTO Pilote (brevet,nom, pseudo,nbHVol,compa)`
	`-> VALUES ('PL-5', 'Lacombe',`
	`'pizza', 5000, '**LUH**');`
	`ERROR **1452** (23000): a child row: a`
	`foreign key constraint fails ('bdutil'.'pilote',`
	`CONSTRAINT 'fk_Pil_compa_Comp'`
	`FOREIGN KEY ('compa')`
	`REFERENCES 'compagnie' ('comp'))`

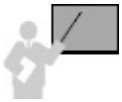
	`mysql> INSERT INTO Pilote (brevet,nom, pseudo,nbHVol,compa)`
	`-> VALUES ('PL-6', **NULL**, 'neant',`
	`1540, 'AF');`
	`ERROR **1048** (23000): Column`

```
                                                'nom' cannot
                                                be null
```

Insertions multilignes

Il est possible d'insérer plusieurs enregistrements dans une table en une seule instruction INSERT. On parle alors d'insertion multiligne. La syntaxe de ce mécanisme est présentée ci-après. Les options sont identiques à l'insertion classique mise à part la directive SELECT (instruction étudiée au chapitre 4) qui décrit l'extraction des données à insérer dans la table.

```
INSERT [LOW_PRIORITY | HIGH_PRIORITY] [IGNORE]
    [INTO] [nomBase.]nomTable [(nomColonne,...)]
    SELECT ...
    [ON DUPLICATE KEY UPDATE colonne=expression,...]
```



On parle d'instruction multiple (*multiple-row statement*) dès qu'une mise à jour (INSERT, UPDATE et DELETE) affecte plus d'une ligne de la table modifiée. Dans le cas d'un ajout, il s'agit d'une instrution INSERT INTO … SELECT … Dans le cas d'une modification ou d'une suppression, il s'agit d'un UPDATE ou DELETE dont la clause WHERE concerne plusieurs lignes.

Supposons l'existence de la table Compagnie_Bis que l'on désire alimenter par toutes les compagnies dont l'adresse est renseignée et qui ne résident pas à Paris.

Tableau 2-6 Insertions multiples

Instructions SQL	Commentaires

`CREATE TABLE Compagnie_Bis`
`(comp VARCHAR(4), nrue INTEGER(3),`
`rue VARCHAR(20), ville VARCHAR(15),`
`nomComp VARCHAR(15), budget DECIMAL (8,2) DEFAULT 0,`
`CONSTRAINT pk_Compagnie_Bis PRIMARY KEY(comp));`
	Création de la table d'accueil.

`INSERT INTO Compagnie_Bis`
`(comp, nrue, rue, ville, nomComp)`
`SELECT comp, nrue, rue, ville, nomComp`
`FROM Compagnie`
`WHERE nrue IS NOT NULL`
`AND rue IS NOT NULL`
`AND ville IS NOT NULL`
`AND NOT(ville = 'Paris');`
	Insertion multiligne dans la table.

Contenu de la table `Compagnie_Bis`

```
+------+------+-----------+-----------+--------------+--------+
| comp | nrue | rue       | ville     | nomComp      | budget |
+------+------+-----------+-----------+--------------+--------+
| LUFT |    9 | Salas     | Munich    | Luftansa     |   0.00 |
| QUAN |    1 | Kangouroo | Sydney    | Quantas      |   0.00 |
| SING |    7 | Camparols | Singapour | Singapore AL |   0.00 |
| SNCM |    3 | P. Paoli  | Bastia    | Corse Air    |   0.00 |
+------+------+-----------+-----------+--------------+--------+
```

Données binaires

Le type `BIT` permet de manipuler des suites variables de bits. Des fonctions sont disponibles pour programmer le « ET », le « OU » exclusif ou inclusif, etc. La table suivante contient deux colonnes de ce type. Notez l'utilisation du préfixe « b » pour initialiser un tel type.

```
CREATE TABLE Registres (nom CHAR(5),numero BIT(2),adresse BIT(16));
INSERT INTO Registres VALUES ('COM2', b'10', b'0000010011110111');
```

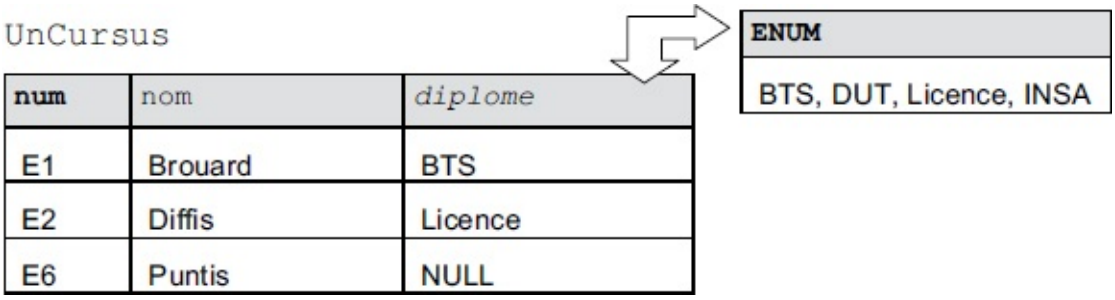
Énumérations

Deux types d'énumérations existent : le type `ENUM` (énumération simple), qui permet de vérifier la présence d'une valeur dans une liste, et le type `SET` (énumération multiple), qui permet de vérifier la présence de plusieurs valeurs dans une liste.

Le type ENUM

Le type `ENUM` se définit par une liste de chaînes de caractères lors de la création de la table. Toute valeur d'une colonne de ce type devra être présente dans cette liste. Supposons que l'on recense 4 diplômes ('`BTS`', '`DUT`', '`Licence`' et '`INSA`'), chacun affecté ou non à un étudiant.

***Figure 2-2** Table avec une colonne `ENUM`*



Le script de création de la table et des insertions valides et incorrectes est le suivant (tableau 2-7). Notez la présence de NULL à la création de l'énumération (autorise l'absence de valeur) et lors d'insertions. Par ailleurs, les parenthèses sont optionnelles pour renseigner la colonne ENUM. La première erreur s'explique par l'absence de valeur de la liste de référence, la seconde démontre qu'une seule valeur d'énumération est permise par colonne.

Tableau 2-7 Insertions avec un ENUM

Table	Insertions (valides et incorrectes)

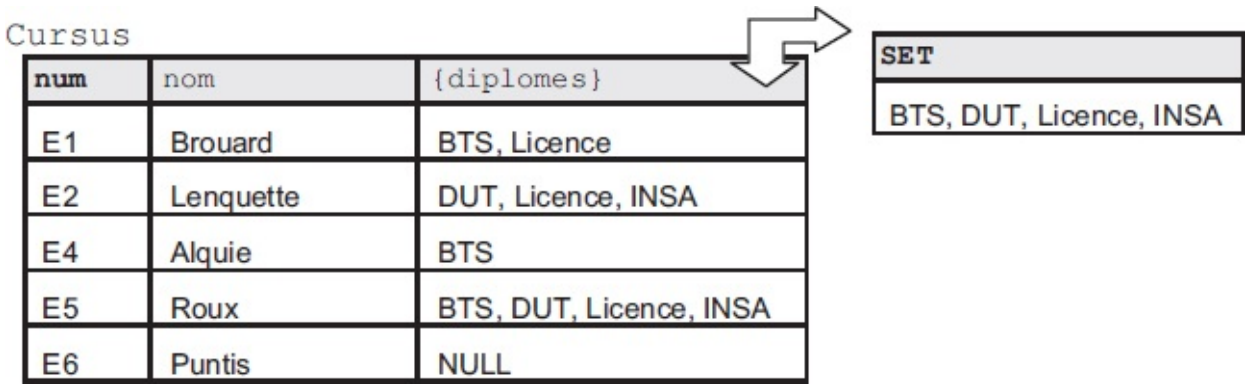
	`INSERT INTO UnCursus (num,nom,diplome)`
	`  VALUES ('E1', 'Brouard', ('BTS'));`
	`INSERT INTO UnCursus (num,nom,diplome)`
	`  VALUES ('E2', 'Diffis', 'Licence');`
	`INSERT INTO UnCursus (num,nom)`
	`  VALUES ('E6','Puntis');`

`CREATE TABLE UnCursus`
` (num CHAR(4), nom VARCHAR(15), diplome`
`ENUM ('BTS','DUT','Licence','INSA') NULL`
` CONSTRAINT pk_UnCursus PRIMARY KEY(num));`
	`mysql> INSERT INTO UnCursus`
	`(num,nom,diplome)`
	`  -> VALUES ('E3', 'Bug', 'MathSup');`
	`ERROR 1265 (01000): Data truncated for`
	`column`
	`'diplome' at row 1`
	`mysql> INSERT INTO UnCursus`
	`(num,nom,diplome)`
	`  -> VALUES ('E4', 'Trop!',`
	`('Licence','BTS'));`
	`ERROR 1241 (21000): Operand should`
	`contain 1`
	`column(s)`

Le type SET

Le type SET se définit aussi par une liste de chaînes de caractères lors de la création de la table. Toute valeur (ou liste de valeurs) d'une colonne de ce type devra être incluse dans la liste de référence. Supposons à présent que chaque étudiant puisse être détenteur d'un ou de plusieurs des 4 diplômes ('BTS', 'DUT', 'Licence' et 'INSA').

***Figure 2-3** Table avec une colonne SET*



Le script de création de la table et des insertions valides et incorrectes est le suivant (tableau 2-8). Notez la présence de NULL à la création de l'énumération multiple (ce qui autorise l'absence de valeur) et lors d'insertions. Par ailleurs, les parenthèses sont optionnelles pour renseigner la colonne SET.

Les doublons sont permis à l'insertion et l'ordre importe peu. Les doublons seront ignorés au niveau du stockage et l'ordre de stockage des différentes valeurs suivra l'ordre alphabétique. L'erreur s'explique par l'absence d'une valeur dans la liste de référence.

Tableau 2-8 Insertion avec un SET

Table	Insertions (valides et incorrectes)
CREATE TABLE Cursus (num CHAR(4), nom VARCHAR(15), diplomes SET ('BTS','DUT','Licence','INSA') NULL, CONSTRAINT pk_Cursus PRIMARY KEY(num));	INSERT INTO Cursus (num,nom,diplomes) VALUES ('E1', 'Brouard', ('BTS,Licence')); INSERT INTO Cursus (num,nom,diplomes) VALUES ('E2', 'Lenquette', 'Licence,INSA,DUT'); INSERT INTO Cursus (num,nom,diplomes) VALUES ('E4', 'Alquie', ('BTS,BTS')); INSERT INTO Cursus (num,nom,diplomes) VALUES ('E5', 'Roux', ('Licence,BTS,INSA,BTS,INSA,INSA,DUT')); INSERT INTO Cursus (num,nom) VALUES ('E6', 'Puntis');
	INSERT INTO Cursus (num,nom,diplomes) VALUES ('E3', 'Bug', ('BTS,INSA,ENAC')); ERROR 1265 (01000): Data truncated for column 'diplomes' at row 1

Ne pas confondre avec les tables de référence

Bien que MySQL assure une certaine cohérence entre les valeurs d'un type ENUM et SET (dont la définition peut varier dans le temps), avec les colonnes de vos tables, ces mécanismes ne peuvent se substituer à l'intégrité référentielle introduite par les clés étrangères (voir la section « Intégrité référentielle »).

Ne confondez donc pas une contrainte « légère » d'un type ENUM et SET avec une relation entre une clé étrangère d'une table et une clé primaire (ou colonne unique) d'une table de référence. Les premiers mécanismes n'ont pas la puissance et la capacité d'évolution que proposent les contraintes référentielles.

Dates et heures

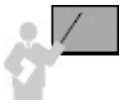
Nous avons décrit au chapitre 1 les caractéristiques générales des types MySQL pour stocker des éléments de type date/heure. Étudions maintenant la manipulation de ces types. Nous verrons que MySQL peut les considérer soit en tant que chaînes de caractères soit comme numériques.

Formats

Concernant les types DATETIME, DATE et TIMESTAMP les formats possibles sont les suivants :

- Chaînes de caractères `'YYYY-MM-DD HH:MM:SS'` ou `'YY-MM-DD HH:MM:SS'` (pour les colonnes DATE `'YYYY-MM-DD'` ou `'YY-MM-DD'`). Tout autre délimiteur est autorisé comme : `'2005.12.31 11%30%45'` (pour les colonnes DATE, `'65.12.31'`, `'65/12/31'`, et `'65@12@31'` sont équivalents et désignent tous le réveillon de l'année 1965).
- Considérés comme chaînes de caractères dans les formats suivants : `'YYYYMMDDHHMMSS'` ou `'YYMMDDHHMMSS'` (pour les DATE `'YYYYMMDD'` ou `'YYMMDD'`) en supposant que la chaîne ait un sens en tant que date. Ainsi `'19650205063000'` est interprété comme le 5 février 1965 à 6 heures et 30 minutes. Par contre '19650255' n'a pas de sens du fait du jour (l'erreur ERROR **1292** (22007) Incorrect date value:… sera retournée).
- Considérés comme numériques dans les formats suivants : YYYYMMDDHHMMSS ou YYMMDDHHMMSS (pour les DATE YYYYMMDD ou YYMMDD) en supposant que le nombre ait un sens en tant que date. Ainsi 19650205063000 est interprété comme le 5 février 1965 à 6 heures et 30 minutes. Par contre 19650205069000 ou 19650205250000 n'ont pas de sens du fait des minutes (6h et 90 min) dans

le premier cas et des heures (25h) pour le second (l'erreur `ERROR 1292 (22007) Incorrect date value:…` sera retournée).



Un bon point pour MySQL qui depuis la version 5.1, selon le mode SQL par défaut (strict), ne transforme plus les dates invalides par `'0000-00-00'` ou `00000000000000` suivant le format original. Les nostalgiques de ce mécanisme trouveront avec le mode SQL `ALLOW_INVALID_DATES` les moyens de tenter le diable (voir le chapitre 6, section « Modes d'exécution SQL »).

Exemple avec `DATE` *et* `DATETIME`

Déclarons la table `Pilote` qui contient deux colonnes de type date/heure : `DATE` et `DATETIME` :

```
CREATE TABLE Pilote
 (brevet VARCHAR(6), nom VARCHAR(20), dateNaiss DATE,
  nbHVol DECIMAL(7,2),dateEmbauche DATETIME, compa VARCHAR(4),
  CONSTRAINT pk_Pilote PRIMARY KEY(brevet));
```

L'insertion du pilote initialise la date de naissance au 5 février 1965, ainsi que celle de l'embauche à la date du moment (heures, minutes, secondes) par la fonction `SYSDATE`.

```
INSERT INTO Pilote (brevet,nom,dateNaiss,nbHVol,dateEmbauche,compa)
  VALUES ('PL-1', 'Christian Soutou', '1965-02-05', 900, SYSDATE(), 'AF');
```

Nous verrons au chapitre 4 comment extraire les années, mois, jours, heures, minutes et secondes. Nous verrons aussi qu'il est possible d'ajouter ou de soustraire des dates entre elles.

Exemple avec `TIME` *et* `YEAR`

Par analogie aux différents formats des dates, les heures (type `TIME` '`HH:MM:SS`' ou '`HHH:MM:SS`') peuvent aussi être manipulées sous la forme de chaînes ou de nombres.

- Chaîne `'D HH:MM:SS.fraction'` avec le nombre de jours (0 à 34) et la fraction de seconde (pas encore opérationnelle), `'HH:MM:SS.fraction'`, `'HH:MM:SS'`, `'HH:MM'`, `'D HH:MM:SS'`, `'D HH:MM'`, `'D HH'`, ou `'SS'`.

- Chaîne sans les délimiteurs sous réserve que la chaîne ait un sens. Ainsi, `'101112'` est considéré comme `'10:11:12'`, mais `'109712'` est incorrect du fait des minutes, il devient `'00:00:00'`.
- Nombre en raisonnant comme les chaînes. Ainsi `101112` est considéré comme `'10:11:12'`, mais `109712` est incorrect du fait des minutes, il devient `'00:00:00'`. Les formats suivants sont aussi corrects : `SS`, `MMSS`, `HHMMSS`.

La table `Pilote` suivante contient la colonne `pasVolDepuis` pour stocker le délai depuis le dernier vol et la colonne `retraite` qui indique l'année de retraite.

```
CREATE TABLE Pilote
 (brevet VARCHAR(6), nom VARCHAR(20), pasVolDepuis TIME, retraite YEAR,
  CONSTRAINT pk_Pilote PRIMARY KEY(brevet));
```

Les insertions suivantes utilisent différents formats. Le premier pilote n'a pas volé depuis 1 jour et 23 heures (47 heures) ; le second depuis 15 heures, 26 minutes 30 secondes ; le troisième depuis 4 jours et 23 heures (119 heures) ; le quatrième depuis 3 heures 27 minutes et 50 secondes, et le dernier se croit plus précis que tous les autres en ajoutant une fraction qui ne sera pas prise en compte au niveau de la base.

```
INSERT INTO Pilote VALUES ('PL-1', 'Hait', '1 23:0:0', '2002');
INSERT INTO Pilote VALUES ('PL-2', 'Crampes', '152630', 2006);
INSERT INTO Pilote VALUES ('PL-3', 'Tuffery', '4 23:00', 05);
INSERT INTO Pilote VALUES ('PL-4', 'Mercier', 032750, '07');
INSERT INTO Pilote VALUES ('PL-5', 'Albaric', '1 23:0:0.457', '01');
```

L'état de la base est le suivant :

```
mysql> SELECT brevet,nom,pasVolDepuis,retraite FROM Pilote;
+--------+---------+--------------+----------+
| brevet | nom     | pasVolDepuis | retraite |
+--------+---------+--------------+----------+
| PL-1   | Hait    | 47:00:00     |     2002 |
| PL-2   | Crampes | 15:26:30     |     2006 |
| PL-3   | Tuffery | 119:00:00    |     2005 |
| PL-4   | Mercier | 03:27:50     |     2007 |
| PL-5   | Albaric | 47:00:00     |     2001 |
+--------+---------+--------------+----------+
```

Si un dépassement se produit au niveau d'une colonne `TIME`, celle-ci est évaluée au maximum ou au minimum ('-850:00:00' et '999:00:00' sont respectivement convertis à '-838:59:59' et '838:59:59').

Exemple avec `TIMESTAMP`

Par défaut, toute colonne du type `TIMESTAMP` est actualisée lors de toute modification de l'enregistrement (la première fois à l'`INSERT`, puis à chaque `UPDATE`, quelle que soit la colonne mise à jour). Déclarons la table `Pilote` qui contient une colonne de ce type.

```
CREATE TABLE Pilote
(brevet VARCHAR(6), nom VARCHAR(20), misaJour TIMESTAMP,
CONSTRAINT pk_Pilote PRIMARY KEY(brevet));
```

L'insertion du pilote suivant initialisera la colonne à la date et heure système.

```
mysql> INSERT INTO Pilote (brevet,nom) VALUES ('PL-1', 'Giaconne');
```

Le résultat vérifie ce mécanisme :

```
mysql> SELECT * FROM Pilote;
+--------+----------+---------------------+
| brevet | nom      | misaJour            |
+--------+----------+---------------------+
| PL-1   | Giaconne | 2013-02-21 16:04:35 |
+--------+----------+---------------------+
```

Par la suite, et à la différence d'un type `DATE` ou `DATETIME`, pour chaque modification de ce pilote, la colonne `misaJour` sera réactualisée avec la date de l'instant de la mise à jour. Même si vous désirez forcer à `NULL` cette colonne avec une instruction `UPDATE`, elle contiendra toujours l'instant de votre vaine tentative !

MySQL 5.6 et ultérieures

Depuis la version 5.6, les options `DEFAULT CURRENT_TIMESTAMP` et `ON UPDATE CURRENT_TIMESTAMP` s'appliquent aussi bien aux colonnes de type `TIMESTAMP` ou `DATETIME`. La table suivante utilise ces deux options sur deux colonnes distinctes.

```
CREATE TABLE Pilote
 (brevet   VARCHAR(6), nom VARCHAR(20),
  ajoute   TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
  misaJour TIMESTAMP DEFAULT 0 ON UPDATE CURRENT_TIMESTAMP,
  CONSTRAINT pk_Pilote PRIMARY KEY(brevet));
```

Ainsi, un ajout se répercute par la mise à jour de la seule colonne dont la valeur par défaut n'est pas nulle.

```
mysql> INSERT INTO Pilote (brevet,nom) VALUES ('PL-1', 'Giaconne');


mysql> SELECT * FROM Pilote;
+--------+----------+---------------------+---------------------+
| brevet | nom      | ajoute              | misaJour            |
+--------+----------+---------------------+---------------------+
| PL-1   | Giaconne | 2013-02-21 16:16:09 | 0000-00-00 00:00:00 |
+--------+----------+---------------------+---------------------+
```

Les modifications ultérieures n'agiront que sur la colonne `misaJour` qui a été configurée à cet effet (option `ON UPDATE…`).

Il est possible d'agir de même avec un type `DATETIME`. Le tableau suivant présente la différence entre ces types au niveau de la valeur par défaut lorsque la colonne est contrainte avec le prédicat `NULL` (ou `NOT NULL`).

Tableau 2-9 Options de l'estampillage

Déclaration de la colonne	Valeur par défaut
TIMESTAMP ON UPDATE CURRENT_TIMESTAMP	0
TIMESTAMP NULL ON UPDATE CURRENT_TIMESTAMP	NULL
DATETIME ON UPDATE CURRENT_TIMESTAMP	NULL
DATETIME NOT NULL ON UPDATE CURRENT_TIMESTAMP	0

Fonctions utiles

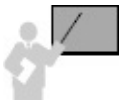
Les fonctions `CURRENT_TIMESTAMP()`, `CURRENT_DATE()` et `CURRENT_TIME()`, `UTC_TIME()`, renseignent sur l'instant, la date, l'heure et l'heure GMT de la session en cours. Il n'est pas nécessaire d'utiliser une table pour afficher une expression dans l'interface de commande. L'exemple suivant montre que la requête a été exécutée le 1er novembre 2005 à 10 heures, 11 minutes et 27 secondes. Le client est sur le fuseau GMT+1h.

```
mysql> SELECT CURRENT_TIMESTAMP(), CURRENT_TIME(), CURRENT_DATE(), UTC_TIME();
+---------------------+---------------+---------------+------------+
| CURRENT_TIMESTAMP() | CURRENT_TIME() | CURRENT_DATE() | UTC_TIME() |
+---------------------+---------------+---------------+------------+
| 2005-11-01 10:11:27 | 10:11:27      | 2005-11-01    | 09:11:27   |
+---------------------+---------------+---------------+------------+
```

Séquences

Bien que « séquence » ne soit pas dans le vocabulaire de MySQL, car le mécanisme qu'il propose n'est pas aussi puissant que celui d'Oracle, MySQL offre la possibilité de générer automatiquement des valeurs numériques. Ces

valeurs sont bien utiles pour composer des clés primaires de tables quand vous ne disposez pas de colonnes adéquates à cet effet. Ce mécanisme répond en grande partie à ce qu'on attendrait d'une séquence.

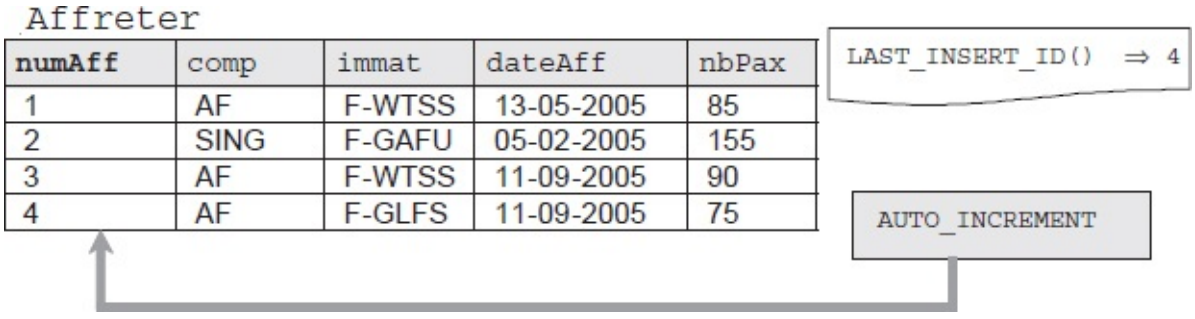


En attendant que MySQL offre peut-être dans une prochaine version un mécanisme plus riche que celui que nous allons étudier, je me permets d'appeler « séquence » une colonne indexée de type entier (`INTEGER`, `SMALLINT`, `TINYINT`, `MEDIUMINT` et `BIGINT`), définie à l'aide de la directive `AUTO_INCREMENT`. Cette colonne est clé primaire, ou unique et non nulle. Une séquence est en général affectée à une table, mais vous pouvez l'utiliser pour plusieurs tables ou variables.

Utilisation en tant que clé primaire

La figure suivante illustre la séquence appliquée à la colonne `numAff` pour initialiser les valeurs de la clé primaire de la table `Affreter`. La fonction `LAST_INSERT_ID()` retourne la dernière valeur de la séquence générée (ici le pas est de 1, la séquence débute par défaut à 1, nous verrons par la suite qu'il est possible d'utiliser une valeur différente).

Figure 2-4 *Séquence appliquée à une clé primaire*



Le tableau suivant décrit la création de la table et sa séquence, ainsi que différentes insertions valides. Notez qu'il n'est pas possible de forcer une valeur pour la séquence.

Tableau 2-10 Séquence pour une clé primaire

Table	Insertions

`CREATE TABLE Affreter`
`(numAff SMALLINT AUTO_INCREMENT,`
`comp CHAR(4), immat CHAR(6),`
`dateAff DATE, nbPax SMALLINT(3),`
`CONSTRAINT pk_Affreter`
`PRIMARY KEY (numAff));`
	`INSERT INTO Affreter`
	`(comp,immat,dateAff,nbPax)`
	`VALUES ('AF','F-WTSS','2005-05-13',85);`
	`INSERT INTO Affreter`
	`(comp,immat,dateAff,nbPax)`
	`VALUES ('SING','F-GAFU','2005-02-05',155);`
	`INSERT INTO Affreter`
	`(numAff,comp,immat,dateAff,nbPax)`
	`VALUES (NULL, 'AF','F-WTSS', '2005-09-11',`
	`90);`
	`INSERT INTO Affreter`
	`VALUES (0,'AF','F-GLFS','2005-09-11',75);`

Modification d'une séquence

La seule modification possible d'une séquence est celle qui consiste à changer la valeur de départ de la séquence (avec `ALTER TABLE`). Seules les valeurs à venir de la séquence modifiée seront changées (heureusement pour les données existantes des tables).

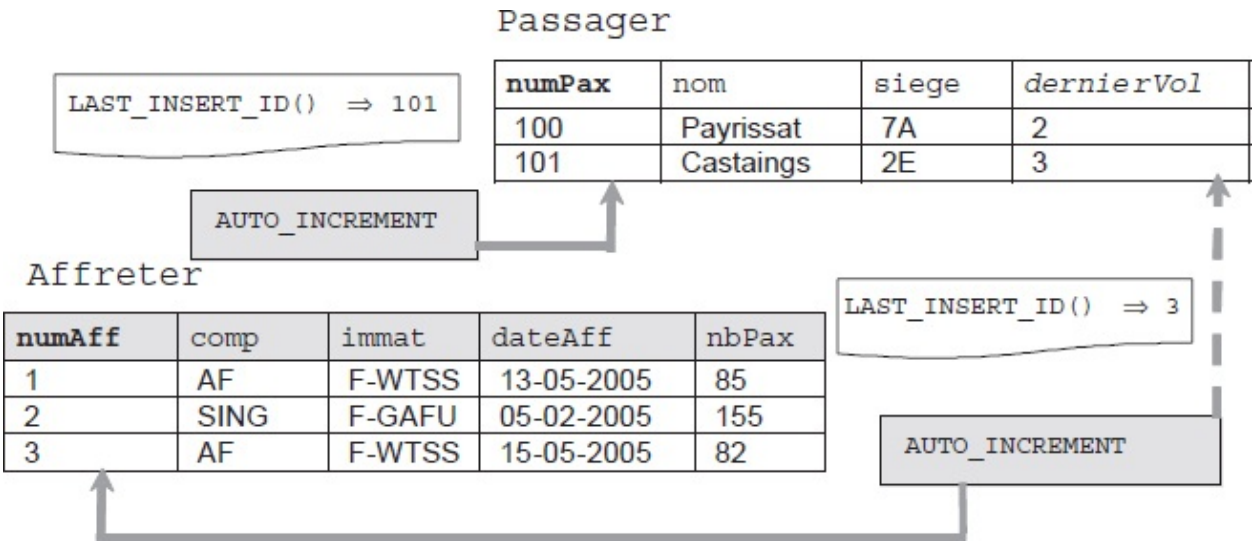
Supposons qu'on désire continuer à insérer des nouveaux affrètements à partir de la valeur 100. Le prochain affrètement sera estimé à 100 et les insertions suivantes prendront en compte le nouveau point de départ tout en laissant intactes les données existantes des tables.

```
ALTER TABLE Affreter AUTO_INCREMENT = 100;
INSERT INTO Affreter  (comp,immat,dateAff,nbPax)
    VALUES ('SING', 'F-NEW', SYSDATE(), 77);
mysql> SELECT * FROM Affreter ;
+--------+------+--------+------------+-------+
| numAff | comp | immat  | dateAff    | nbPax |
+--------+------+--------+------------+-------+
|      1 | AF   | F-WTSS | 2005-05-13 |    85 |
|      2 | SING | F-GAFU | 2005-02-05 |   155 |
|      3 | AF   | F-WTSS | 2005-09-11 |    90 |
|      4 | AF   | F-GLFS | 2005-09-11 |    75 |
|    100 | SING | F-NEW  | 2005-11-01 |    77 |
+--------+------+--------+------------+-------+
```

Utilisation en tant que clé étrangère

Créons deux séquences qui vont permettre de donner leur valeur aux clés primaires des deux tables illustrées à la figure suivante (les affrètements commencent à 1, les passagers à 100). Servons-nous aussi de la séquence de la table `Affreter` pour indiquer le dernier vol de chaque passager. La section « Intégrité référentielle » détaille les mécanismes relatifs aux clés étrangères.

Figure 2-5 *Séquence appliquée à une clé étrangère*



Le script SQL de définition et de manipulation des données est indiqué ci-après. La valeur de départ d'une séquence peut être définie à la fin de l'ordre CREATE TABLE. Notez également l'utilisation de la fonction LAST_INSERT_ID dans les insertions pour récupérer la valeur de la clé primaire.

Tableau 2-11 Séquence pour une clé étrangère

Tables	Insertions

`CREATE TABLE Affreter`
`(numAff SMALLINT AUTO_INCREMENT,`
`comp CHAR(4), immat CHAR(6),`
`dateAff DATE,nbPax SMALLINT(3),`
`CONSTRAINT pk_Affreter`
`PRIMARY KEY (numAff));`
`CREATE TABLE Passager`
`(numPax SMALLINT AUTO_INCREMENT,`
`nom CHAR(15), siege CHAR(4),`
`dernierVol SMALLINT,`
`CONSTRAINT pk_Passager`
`PRIMARY KEY(numPax),`
`CONSTRAINT fk_Pax_vol_Affreter`
`FOREIGN KEY (dernierVol)`
`REFERENCES Affreter(numAff))`
`AUTO_INCREMENT = 100;`
	`INSERT INTO Affreter`
	`(comp,immat,dateAff,nbPax) VALUES`
	`('AF','F-WTSS','2005-05-13',85);`
	`INSERT INTO Affreter`
	`(comp,immat,dateAff,nbPax) VALUES`
	`('SING','F-GAFU','2005-02-05',155);`
	`INSERT INTO Passager VALUES`
	`(NULL,'Payrissat','7A',LAST_INSERT_ID());`
	`INSERT INTO Affreter VALUES`
	`(NULL,'AF','F-WTSS','2005-05-15',82);`
	`INSERT INTO Passager VALUES`
	`(NULL,'Castaings','2E',LAST_INSERT_ID());`

Le mécanisme d'auto-incrémentation est très utilisé en production. En effet, il est pratique et optimal que chaque table dispose d'une clé primaire sous la forme d'une séquence (index compact et efficace du fait que les lignes sont

proches dans les blocs de données). On parle de clé artificielle.

Par ailleurs, il est souhaitable que chaque table dispose également d'une clé métier (ayant une sémantique forte comme le code d'un produit ou le numéro de brevet d'un pilote). Cette dernière peut être implémentée sous la forme d'une contrainte unique afin de bénéficier d'un index.

Modifications de colonnes

L'instruction UPDATE permet la mise à jour des colonnes d'une table. Pour pouvoir modifier des enregistrements d'une table, il faut que cette dernière soit dans votre base ou que vous ayez reçu le privilège UPDATE sur la table.

Syntaxe [UPDATE]

La syntaxe simplifiée de l'instruction UPDATE est la suivante :

```
UPDATE [LOW_PRIORITY] [IGNORE] [nomBase.] nomTable
SET col_name1=expr1 [, col_name2=expr2 ...]
      SET     colonne1 = expression1 | (requête_SELECT) | DEFAULT
            [,colonne2 = expression2...]
      [WHERE (condition)]
      [ORDER BY listeColonnes]
      [LIMIT nbreLimite]
```

- LOW_PRIORITY diffère l'exécution de la commande tant qu'il existe un client qui accède à la table. Cela ne concerne que les moteurs de stockage qui verrouillent au niveau table (MyISAM, MEMORY et MERGE).
- IGNORE signifie que les éventuelles erreurs déclenchées suite aux modifications seront considérées en tant que *warning*s.
- La clause SET actualise une colonne en lui affectant une expression (valeur, valeur par défaut, calcul ou résultat d'une requête).
- La condition du WHERE filtre les lignes à mettre à jour dans la table. Si aucune condition n'est précisée, tous les enregistrements seront actualisés. Si la condition ne filtre aucune ligne, aucune mise à jour ne sera réalisée.
- ORDER BY indique l'ordre de modification des colonnes.
- LIMIT spécifie le nombre maximum d'enregistrements à changer (par ordre de clé primaire croissante).

Modification d’une colonne

Modifions la compagnie de code 'AN1' en affectant la valeur 50 à la colonne `nrue`.

```
UPDATE Compagnie SET nrue = 50 WHERE comp = 'AN1';
```

Modification de plusieurs colonnes

Modifions la compagnie de code 'AN2' en affectant simultanément la valeur 14 à la colonne `nrue` et la valeur par défaut ('`Paris`') à la colonne `ville`.

```
UPDATE Compagnie SET nrue = 14, ville = DEFAULT WHERE comp = 'AN2';
```

La table `Compagnie` contient à présent les données suivantes :

Figure 2-6 *Table après les modifications*

Compagnie — Modification 1

comp	nrue	rue	ville	nomComp
SING	7	Camparols	Singapour	Singapore AL
AF	10	Gambetta	Paris	Air France
AN1	**50**	Hoche	Blagnac	Air Nul1
AC	8	Champs Elysées	Paris	Castanet Air
AN2	**14**	Foch	**Paris**	Air Nul2

Modifications 2

Modification de plusieurs enregistrements

Modifions les deux premières compagnies (par ordre de clé primaire, ici 'AC' et 'AF') en affectant la valeur 'Toulouse' à la colonne `ville`.

```
UPDATE Compagnie SET ville = 'Toulouse' LIMIT 2;
```

Ne pas respecter les contraintes

Il faut, comme pour les insertions, respecter les contraintes qui existent au niveau des colonnes. Dans le cas inverse, une erreur est renvoyée (le nom de la contrainte apparaît) et la mise à jour n’est pas effectuée.

Tableau 2-12 Table, données et contraintes

Données	Table et contraintes

Figure 2-7 *Données*

Pilote

brevet	nom	nbHVol	compa
PL-1	Ferrage	450	AF
PL-2	Guilbaud	900	AF
PL-3	Lacombe	1000	SING

```
CREATE TABLE Pilote
(brevet CHAR(6), nom VARCHAR(15) NOT
NULL,
 nbHVol DECIMAL(7,2), compa CHAR(4),
 CONSTRAINT pk_Pilote
   PRIMARY KEY(brevet),
CONSTRAINT ck_nbHVol
   CHECK (nbHVol BETWEEN 0 AND 20000),
CONSTRAINT un_nom UNIQUE(nom),
CONSTRAINT fk_Pil_compa_Comp
   FOREIGN KEY (compa)
    REFERENCES Compagnie(comp));
```

À partir de la table `Pilote`, le tableau suivant décrit des modifications (certaines ne vérifient pas de contraintes). La mise à jour d'une clé étrangère est possible si elle n'est pas référencée par une clé primaire (voir la section « Intégrité référentielle »).

Prenez garde aux mises à `NULL` par `UPDATE` de colonnes déclarées `NOT NULL` au sein de sessions où le mode SQL n'est pas strict (exemple : `ALLOW_INVALID_DATES`). Vous aurez des mauvaises surprises (mise à `NULL` impossible à l'insertion, mais autorisée en modification…).

Tableau 2-13 Modifications

Vérifiant les contraintes (sauf une !)	Ne vérifiant pas les contraintes

```
--Modification d'une clé étrangère
UPDATE Pilote SET compa = 'SING'
  WHERE brevet = 'PL-2';
-- Modification d'une clé primaire
UPDATE Pilote SET brevet = 'PL3bis'
  WHERE brevet = 'PL-3';
--Passe outre la contrainte CHECK !
UPDATE Pilote SET nbHVol= 30000
  WHERE brevet = 'PL-1';
```

Figure 2-8 *Après modifications*

```
mysql> UPDATE Pilote SET brevet='PL-2'
       WHERE brevet='PL-1';
ERROR 1062 (23000): Duplicate entry
'PL-2' for key 1
mysql> UPDATE Pilote SET nom = NULL
       WHERE brevet = 'PL-1';
ERROR 1048 (22004): Column set to
default value; NULL supplied to NOT
NULL column 'nom' at row 1
mysql> UPDATE Pilote SET nom='Lacombe'
WHERE brevet = 'PL-1';
```

Pilote

brevet	nom	nbHVol	compa
PL-1	Ferrage	**30000**	AF
PL-2	Guilbaud	900	**SING**
PL3bis	Lacombe	1000	SING

```
ERROR 1062 (23000): Duplicate entry
'Paul Soutou' for key 2
```

```
mysql> UPDATE Pilote SET compa='RIEN'
       WHERE brevet = 'PL-1';
ERROR 1452 (23000): Cannot add or update
a child row: a foreign key constraint
fails (`bdsoutou/pilote`, CONSTRAINT
`fk_Pil_compa_Comp` FOREIGN KEY
(`compa`) REFERENCES `compagnie`
(`comp`))
```

Restrictions

Il n'est pas possible d'utiliser la table qui est modifiée par UPDATE dans une sous-requête de la clause SET.

Considérons la mise à jour du pilote de code 'PL-2' en lui affectant la moitié des heures de vol du pilote de code 'PL-1'. L'instruction suivante s'applique :

```
mysql> UPDATE Pilote
         SET nbHVol= (SELECT nbHVol/2 FROM Pilote WHERE brevet='PL-1'
         WHERE brevet='PL-2';
ERROR 1093 (HY000): You can't specify target table 'Pilote' for
update in FROM clause
```

Dates et intervalles

Le tableau suivant résume les principales opérations possibles entre des colonnes de type date-heure.

Tableau 2-14 Opérations entre colonnes date-heure

Opérande 1	Opérateur	Opérande 2	Résultat
DATE | DATETIME	+ ou -	Interval	DATE | DATETIME
DATE | DATETIME	+ ou -	INTEGER	DATE | DATETIME
	+ ou -		

TIME		TIME	TIME
TIME	+ ou -	INTEGER	TIME

Considérons la table suivante :

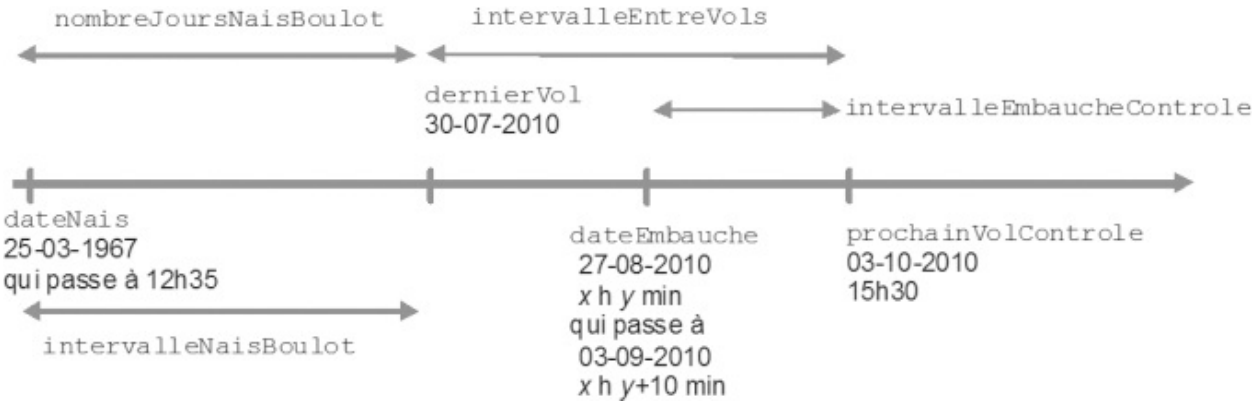
```
CREATE TABLE Pilote
 (brevet VARCHAR(6), nom VARCHAR(20), dateNaiss DATETIME, dernierVol DATE,
  dateEmbauche DATETIME, prochainVolControle DATETIME,
  nombreJoursNaisBoulot INTEGER(5),
  intervalleNaisBoulot Decimal (20,6), intervalleVolExterieur
  Decimal (10,6),
  intervalleEntreVols Decimal (10,6),  intervalleEmbaucheControle
  TIME,
  compa VARCHAR(4), CONSTRAINT pk_Pilote PRIMARY KEY(brevet));
```

À l'insertion du pilote 'Thierry Albaric' de la compagnie de code 'AF', initialisons sa date de naissance (25 mars 1967), la date de son dernier vol (30 juilllet 2010), sa date d'embauche (celle du jour) et la date de son prochain contrôle en vol (3 octobre 2010 à 15h30).

```
INSERT INTO Pilote
(brevet,nom,dateNaiss,dernierVol,dateEmbauche,prochainVolControle,
nombreJoursNaisBoulot,intervalleNaisBoulot,intervalleVolExterieur,
  intervalleEntreVols,intervalleEmbaucheControle,compa)
VALUES
('PL-1', 'Thierry Albaric','1967-03-25','2010-07-30', SYSDATE(),
  '2010-10-03 15:30:00', NULL, NULL, NULL, NULL, NULL, 'AF');
```

Les mises à jour par UPDATE sur cet enregistrement vont consister, sur la base de ces quatre dates, à calculer les intervalles illustrés à la figure suivante :

***Figure 2-9** Intervalles à calculer*



Modification d'une heure

On modifie une date en précisant une heure via la fonction en rajoutant le

format 'HH:MM:SS' quels que soient les délimiteurs (ici « : »).

```
UPDATE Pilote SET dateNaiss = '1967-03-25 12:35:00'
              WHERE brevet = 'PL-1';
```

Ajout d'un délai

On modifie la date d'embauche de 10 minutes après la semaine prochaine. L'ajout d'un intervalle s'opère par la fonction DATE_ADD couplée à la directive DAY_MINUTE qui permet de spécifier un jour, une heure et une minute. Ainsi la semaine à ajouter correspond au « 7 » du paramètre, de même pour les 10 minutes.

```
UPDATE Pilote
  SET dateEmbauche = DATE_ADD(dateEmbauche,INTERVAL '7 0:10' DAY_MINUTE)
  WHERE brevet     = 'PL-1';
```

Différence entre deux dates

La différence entre deux dates peut se programmer à l'aide de la fonction DATEDIFF qui renvoie un entier correspondant au nombre de jours séparant les deux dates.

```
UPDATE Pilote
   SET nombreJoursNaisBoulot = DATEDIFF(dateEmbauche,dateNaiss)
   WHERE brevet              = 'PL-1';
```

Cette même différence au format d'un intervalle plus précis (nombre de jours décimaux) requiert l'utilisation de la fonction TIMESTAMPDIFF(*intervalle*,*datetime1*,*datetime2*) que nous étudierons au chapitre 4 (il existe bien sûr d'autres possibilités de programmation). Cette fonction effectue la différence entre deux dates suivant un format d'intervalle donné (ici on obtient des secondes qu'on divise par (24*3600) pour convertir en jours).

```
UPDATE Pilote SET
   intervalleNaisBoulot =
       TIMESTAMPDIFF(SECOND,dateNaiss,dateEmbauche)/(24*3600) ,
   intervalleEntreVols =
       TIMESTAMPDIFF(SECOND,dernierVol,prochainVolControle)/(24*3600),
   intervalleVolExterieur =
       TIMESTAMPDIFF(SECOND,dernierVol,dateEmbauche)/(24*3600)
 WHERE brevet = 'PL-1';
```

Différence entre deux intervalles

La différence entre deux décimaux renvoie un décimal qu'on convertit en

format `TIME` (s'il est inférieur à 839 h, soit 34,95 jours) par la fonction `SEC_TO_TIME` après l'avoir changé en secondes (multiplié par 24*3600).

```
UPDATE Pilote SET
  intervalleEmbaucheControle =
      SEC_TO_TIME((intervalleEntreVols - intervalleVolExterieur)*24*3600)
 WHERE brevet = 'PL-1';
```

La ligne contient désormais les informations suivantes. Les données en gras correspondent aux mises à jour. On trouve qu'il a fallu 15 868 jours pour que ce pilote soit embauché depuis sa naissance. 35,41 jours séparent le dernier vol du pilote du moment de son embauche. 65,64 jours séparent son dernier vol de son prochain contrôle en vol. La différence entre ces deux délais est de 725 heures, 29 minutes et 6 secondes.

Figure 2-10 *Ligne modifiée par des calculs de dates*

Pilote

brevet	nom	dateNaiss	dernierVol	dateEmbauche	prochainVolControle
PL-1	Thierry Albaric	*1967-03-25* 1967-03-25 **12:35:00**	2010-07-30	*2010-08-27 15:27:42* **2010-09-03 15:37:42**	2010-10-03 15:30:00

nombreJoursNaisBoulot	intervalleNaisBoulot	intervalleVolExterieur
15868	**15867.89...**	**35.41...**

intervalleEntreVols	intervalleEmbaucheControle	compa
65.64...	**725:29:06**	AF

Nous verrons au chapitre 4, comment convertir en jours, heures, minutes et secondes un décimal de grande taille.

Fonctions utiles

Les fonctions suivantes vous seront d'un grand secours pour manipuler des dates et des intervalles.

- `DATE_FORMAT(`*`date`*`,`*`format`*`)` convertit une date (heure) suivant un certain format.
- `EXTRACT(`*`type`* `FROM` *`date`*`)` extrait une partie donnée d'une date (heure).
- `FROM_DAYS(`*`n`*`)` retourne une date à partir d'un entier (le 1/1/0001 correspond à 366) ; `UNIX_TIMESTAMP(`*`date`*`)` retourne le nombre de secondes qui se sont

écoulées depuis le 1er janvier 1970 jusqu'à la date (heure) en paramètre.

- `GET_FORMAT(DATE|TIME|DATETIME,'EUR'|'USA'|'JIS'|'ISO'|'INTERNAL')` retourne un format de date (heure).
- `SEC_TO_TIME(secondes)` convertit un nombre en un type `TIME` et son inverse `TIME_TO_SEC(time)`.
- `STR_TO_DATE(chaine,format)` convertit une chaîne en date (heure) suivant un certain format.
- `TIME(expression)`extrait d'une date (heure) un type `TIME`.
- `TIME_FORMAT(time,format)` convertit un intervalle suivant un certain format.

Les tableaux suivants présentent quelques exemples d'utilisation de ces fonctions :

Tableau 2-15 Quelques formats pour `DATE_FORMAT` et `STR_TO_DATE`

Expression	Résultat	Commentaire
`DATE_FORMAT(SYSDATE(),'%j')`	`306`	Ce n'est pas la vitesse de Daniel dans *Taxi2*, mais le numéro du jour de l'année (ici il s'agit du 2 novembre 2005).
`DATE_FORMAT(dateNaiss,'%W en %M %X')`	`Saturday en March 1967`	Affichage des libellés des jours et des mois en anglais par défaut.
`STR_TO_DATE('11/09/2005 15:37:42','%d/%m/%Y %H:%i:%s')`	`2005-09-11 15:37:42`	Conversion d'une chaîne typée date française au format `DATETIME`.

Tableau 2-16 Utilisation de `EXTRACT` et `UNIX_TIMESTAMP`

Expression	Résultat	Commentaire
`EXTRACT(DAY FROM dateEmbauche)`	`9`	Extraction du jour contenu dans la colonne.
`EXTRACT(MONTH FROM dateNaiss)`	`3`	Extraction du mois contenu dans la colonne.

`UNIX_TIMESTAMP(SYSDATE())/(24*3600)`	`13089.8850`	Le 2 novembre 2005 au soir, 13 089 jours et des poussières s'étaient écoulés depuis 1970.

Remplacement d'un enregistrement

L'instruction REPLACE consiste, comme son nom l'indique, à remplacer un enregistrement dans sa totalité (toutes ses colonnes). Il faut avoir les privilèges INSERT et DELETE sur la table. C'est selon la valeur de la clé primaire ou celle d'un index unique que l'enregistrement sera remplacé.

Si la table ne dispose pas d'une contrainte PRIMARY KEY ou UNIQUE, l'utilisation de REPLACE n'a pas de sens et devient équivalente à INSERT.

La syntaxe simplifiée de l'instruction REPLACE est la suivante :

```
REPLACE [LOW_PRIORITY | DELAYED]
      [INTO] [nomBase.] nomTable [(colonne1,...)]
      VALUES ({expression1 | DEFAULT},...) [,(...),...]
```

- LOW_PRIORITY et DELAYED ont la même signification que pour INSERT et UPDATE.
- VALUES contient les valeurs de remplacement.

L'instruction suivante remplace l'enregistrement relatif à la compagnie de code 'AN1' (voir figure 2-6) :

```
REPLACE INTO Compagnie
         VALUES ('AN1', 24, 'Salas', 'Ramonville', 'Air RENATO');
```

Suppressions d'enregistrements

Les instructions DELETE et TRUNCATE permettent de supprimer un ou plusieurs enregistrements d'une table. Pour pouvoir supprimer des enregistrements dans une table, il faut que cette dernière soit dans votre base ou que vous ayez reçu le privilège DELETE sur la table.

Instruction DELETE

La syntaxe simplifiée de l'instruction DELETE est la suivante :

```
DELETE [LOW_PRIORITY] [QUICK] [IGNORE] FROM [nomBase.] nomTable
      [WHERE (condition)]
      [ORDER BY listeColonnes]
      [LIMIT nbreLimite]
```

- LOW_PRIORITY, IGNORE et LIMIT ont la même signification que pour les instructions INSERT et UPDATE.
- QUICK (pour les tables de type MyISAM) ne met pas à jour les index associés pour accélérer le traitement.
- La condition du WHERE sélectionne les lignes à supprimer dans la table. Si aucune condition n'est précisée, toutes les lignes seront détruites. Si la condition ne sélectionne aucune ligne, aucun enregistrement ne sera supprimé.
- ORDER BY réalise un tri des enregistrements qui seront effacés dans cet ordre.

Détaillons les possibilités de cette instruction en considérant les différentes tables précédemment définies. La première commande supprime tous les pilotes de la compagnie de code 'AF', la seconde, avec une autre écriture, détruit la compagnie de code 'AF'.

```
DELETE FROM Pilote WHERE compa = 'AF';

DELETE FROM Compagnie WHERE comp = 'AF';
```

Tentons de supprimer une compagnie qui est référencée par un pilote à l'aide d'une clé étrangère. Il s'affiche une erreur qui sera expliquée dans la section « Intégrité référentielle ».

```
mysql> DELETE FROM Compagnie WHERE comp = 'SING';
ERROR 1451 (23000): Cannot delete or update a parent row: a foreign key
constraint fails (`bdsoutou/pilote`, CONSTRAINT `fk_Pil_compa_Comp`
FOREIGN KEY (`compa`) REFERENCES `compagnie` (`comp`))
```

Détruisons enfin les deux premières compagnies (triées par ordre croissant de clé, ici 'AC' et 'AN1') qui ne sont référencées par aucun pilote.

```
DELETE FROM Compagnie LIMIT 2;
```

Instruction TRUNCATE

La commande TRUNCATE est une extension de SQL qui a été proposée par Oracle et reprise par MySQL. Cette commande supprime tous les enregistrements d'une table et libère éventuellement l'espace de stockage utilisé par la table. La syntaxe est la suivante :

```
TRUNCATE [TABLE] [nomBase.] nomTable;
```

Avec le moteur InnoDB, l'opération est programmée en DELETE. Pour les autres moteurs, l'opération diffère de DELETE de la manière suivante :

- La table est supprimée (DROP) puis recréée (CREATE), ce qui est plus rapide que de détruire les enregistrements un à un.
- L'opération peut être interrompue si une transaction active utilise la table (ou si un verrou est posé).
- Le nombre d'enregistrements supprimés n'est pas retourné.
- L'éventuelle dernière valeur d'une colonne AUTO_INCREMENT n'est pas mémorisée.

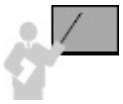
Il n'est pas possible de « tronquer » une table qui est référencée par des clés étrangères actives. La solution consiste à désactiver les contraintes puis à tronquer la table.

Intégrité référentielle

L'intégrité référentielle forme le cœur de la cohérence d'une base de données relationnelle. Cette intégrité est fondée sur la relation entre clés étrangères et clés primaires (ou candidates : colonnes indexées uniques et non nulles) qui permettent de programmer des règles de gestion (exemple : l'affrètement d'un vol doit se faire par une compagnie et pour un avion tous deux existant dans la base de données). Ce faisant, la plupart des contrôles côté client (interface) sont ainsi déportés côté serveur.

Pour les règles de gestion plus complexes (exemple : l'affrètement d'un avion doit se faire par une compagnie qui a embauché au moins quinze pilotes dans les six derniers mois), il faudra programmer un déclencheur (décrits au

chapitre 7). Il faut savoir que les déclencheurs sont plus pénalisants que des contraintes dans un mode transactionnel.



La contrainte référentielle concerne toujours deux tables – une table « père » aussi dite « maître » (*parent*/*referenced*) et une table « fils » (*child*/*dependent*) – possédant une ou plusieurs colonnes en commun. Pour la table « père », ces colonnes composent la clé primaire (ou candidate avec un index unique). Pour la table « fils », ces colonnes composent une clé étrangère.

Syntaxe

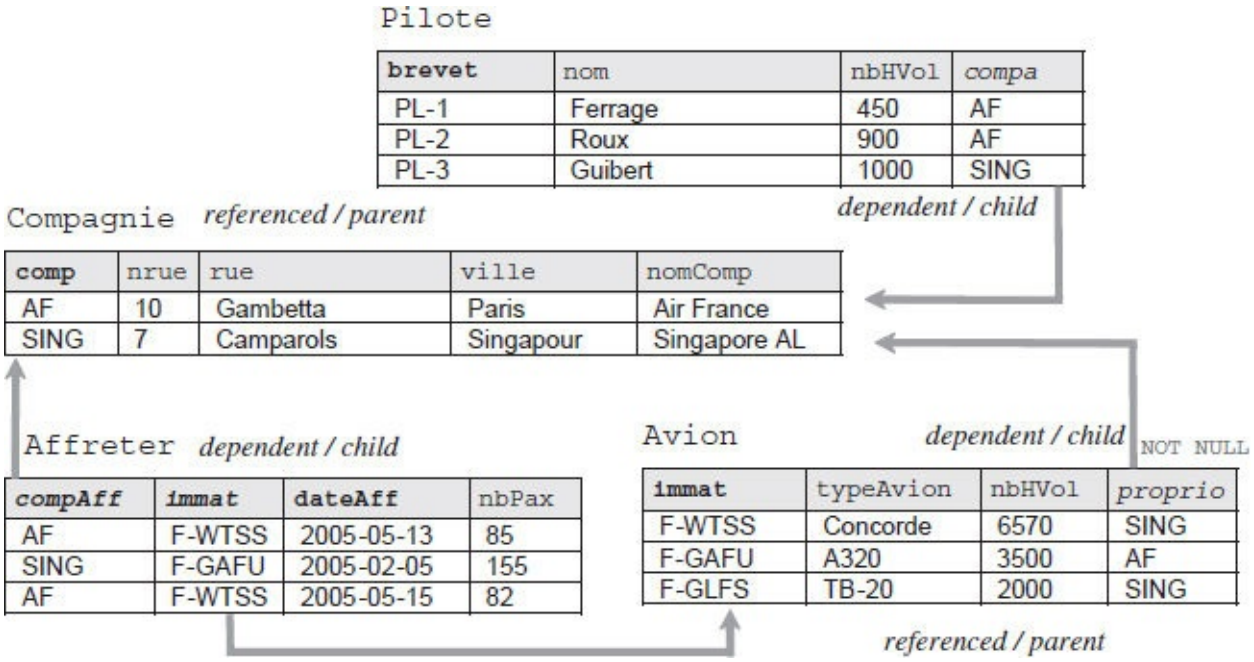
C'est seulement dans sa version 3.23.44 en 2002 (dix ans après Oracle), que MySQL a inclus dans son offre les contraintes référentielles pour les tables `InnoDB`. L'intégrité référentielle se programme dans la table « fils » par la contrainte suivante. Il est conseillé de nommer la contrainte, sinon MySQL s'en charge. Si la clé étrangère n'est pas déjà indexée, MySQL s'en charge aussi. Les deux tables ne doivent pas être temporaires.

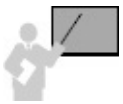
```
[CONSTRAINT nomContrainte] FOREIGN KEY [id] (listeColonneEnfant)
    REFERENCES nomTable (listeColonneParent)
    [ON DELETE {RESTRICT | CASCADE | SET NULL | NO ACTION}]
    [ON UPDATE {RESTRICT | CASCADE | SET NULL | NO ACTION}]
```

Cohérences assurées

L'exemple suivant illustre quatre contraintes référentielles. Une table peut être « père » pour une contrainte et « fils » pour une autre (c'est le cas de la table `Avion`).

Figure 2-11 *Intégrité référentielle*





Deux types de problèmes sont automatiquement résolus par MySQL pour assurer l'intégrité référentielle :

- La cohérence du « fils » vers le « père » : on ne doit pas pouvoir insérer un enregistrement « fils » (ou modifier sa clé étrangère) rattaché à un enregistrement « père » inexistant. Il est cependant possible d'insérer un « fils » (ou de modifier sa clé étrangère) sans rattacher d'enregistrement « père », à la condition qu'il n'existe pas de contrainte NOT NULL au niveau de la clé étrangère.
- La cohérence du « père » vers le « fils » : on ne doit pas pouvoir supprimer un enregistrement « père » si un enregistrement « fils » y est encore rattaché. Il est possible de supprimer les « fils » associés (DELETE CASCADE), d'affecter la valeur nulle aux clés étrangères des « fils » associés (DELETE SET NULL) ou de répercuter une modification de la clé primaire du père (UPDATE CASCADE et UPDATE SET NULL).

Déclarons à présent ces contraintes sous MySQL en détaillant les options disponibles.

Contraintes côté « père »

Le tableau suivant illustre les deux possibilités (clé primaire ou candidate) dans le cas de la table `Compagnie`. Notons que pour le cas de la clé candidate, une clé primaire peut être définie par ailleurs (sur `nomComp` par exemple).

Tableau 2-17 Écritures des contraintes de la table « père »

Clé primaire	Clé candidate

`CREATE TABLE Compagnie`
`(comp CHAR(4), nrue INTEGER(3),`
`rue CHAR(20), ville CHAR(15),`
`nomComp CHAR(15),`
`CONSTRAINT pk_Compagnie PRIMARY KEY(comp));`
	`CREATE TABLE Compagnie`
	`(comp CHAR(4) NOT NULL, nrue INTEGER(3),`
	`rue CHAR(20), ville CHAR(15),`
	`nomComp CHAR(15),`
	`CONSTRAINT un_Compagnie UNIQUE(comp),`
	`CONSTRAINT pk_Compagnie PRIMARY KEY(nomComp));`

Contraintes côté « fils »

Indépendamment de l'écriture de la table « père », plusieurs écritures sont possibles au niveau de la table « fils » selon qu'on créé les index ou qu'on laisse MySQL le faire, et selon qu'on nomme ou pas la contrainte de clé étrangère.

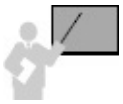
La première écriture nomme la contrainte, mais ne précise rien à propos de l'index. La deuxième ne nomme pas la contrainte, mais définit l'index (sans option toutefois). L'écriture à adopter est un mélange des deux, à savoir nommer les contraintes et décrire les index.

Tableau 2-18 Écritures des contraintes de la table « fils »

Contrainte nommée sans index	Contrainte pas nommée et index

`CREATE TABLE Pilote`
`(brevet CHAR(6), nom CHAR(15),`
`nbHVol DECIMAL(7,2), compa CHAR(4),`
`CONSTRAINT pk_Pilote`
`PRIMARY KEY(brevet),`
`CONSTRAINT fk_Pil_compa_Comp`
`FOREIGN KEY (compa)`
`REFERENCES Compagnie(comp));`
	`CREATE TABLE Pilote`
	`(brevet CHAR(6), nom CHAR(15),`
	`nbHVol DECIMAL(7,2), compa CHAR(4),`
	`CONSTRAINT pk_Pilote`
	`PRIMARY KEY(brevet),`
	`INDEX (compa),`
	`FOREIGN KEY (compa) REFERENCES Compagnie(comp));`

Clés composites et nulles



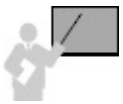
Les clés étrangères ou primaires peuvent être définies sur plusieurs colonnes (16 au maximum), on parle de *composite keys*.

Des clés étrangères peuvent être nulles (si elles ne font pas partie d'une clé primaire) si aucune contrainte NOT NULL n'est déclarée.

Décrivons à présent le script SQL qui convient à notre exemple (la syntaxe de création des deux premières tables a été discutée plus haut) et étudions ensuite les mécanismes programmés par ces contraintes. Décidons qu'un avion aura toujours un propriétaire (NOT NULL).

```
CREATE TABLE Avion
 (immat CHAR(6), typeAvion CHAR(15), nbhVol DECIMAL(10,2),
  proprio CHAR(4) NOT NULL, CONSTRAINT pk_Avion PRIMARY KEY(immat),
  INDEX (proprio),
  CONSTRAINT fk_Avion_comp_Compag
      FOREIGN KEY(proprio) REFERENCES Compagnie(comp));
CREATE TABLE Affreter
 (compAff CHAR(4), immat CHAR(6), dateAff DATE, nbPax INTEGER(3),
  CONSTRAINT pk_Affreter PRIMARY KEY (compAff, immat, dateAff),
  INDEX (immat),
  CONSTRAINT fk_Aff_na_Avion
      FOREIGN KEY(immat) REFERENCES Avion(immat),
INDEX (compAff),
CONSTRAINT fk_Aff_comp_Compag
     FOREIGN KEY(compAff) REFERENCES Compagnie(comp));
```

Cohérence du fils vers le père



Si la clé étrangère est déclarée NOT NULL, l'insertion d'un enregistrement « fils » n'est possible que s'il est rattaché à un enregistrement « père » existant. Dans le cas inverse, l'insertion d'un enregistrement « fils » rattaché à aucun « père » est possible.

Le tableau suivant décrit des insertions correctes et une incorrecte. Le message d'erreur est ici en anglais (il y est question de ne pouvoir ajouter un enregistrement « fils »).

Tableau 2-19 Insertions correctes et incorrectes

Insertions correctes	Insertion incorrecte

`-- fils avec père`
`INSERT INTO Pilote VALUES`
`('PL-1', 'Ferrage', 1000, 'SING');`
`INSERT INTO Avion VALUES`
`('F-WTSS', 'Concorde', 6570, 'SING');`
`INSERT INTO Affreter VALUES`
`('AF', 'F-WTSS', '15-05-2003', 82)`
`-- fils sans père`
`INSERT INTO Pilote VALUES`
`('PL-4', 'Dupin', 0, NULL);`
	`-- avec père inconnu`
	`mysql> INSERT INTO Pilote VALUES`
	`('PL-5', 'Pb de Compagnie', 0, '?');`
	`ERROR 1452 (23000): Cannot add or update a`
	`child row: a foreign key constraint fails (`bdsoutou/pilote`, CONSTRAINT `fk_Pil_compa_Comp` FOREIGN KEY (`compa`) REFERENCES `compagnie` (`comp`))`

Pour insérer un affrètement, il faut donc avoir ajouté au préalable au moins une compagnie et un avion. Le chargement de la base de données est conditionné par la hiérarchie des contraintes référentielles. Ici il faut insérer d'abord les compagnies, puis les pilotes (ou les avions), enfin les affrètements.

Il suffit de relire le script de création de vos tables pour en déduire l'ordre d'insertion des enregistrements.

Cohérence du père vers le fils

En fonction des options choisies au niveau de la contrainte référentielle se trouvant dans la table « fils » :

```
CONSTRAINT nomContrainte FOREIGN KEY … REFERENCES …
    [ON DELETE {RESTRICT | CASCADE | SET NULL | NO ACTION}]
    [ON UPDATE {RESTRICT | CASCADE | SET NULL | NO ACTION}]
```

plusieurs scénarios sont possibles pour assurer la cohérence de la table « père » vers la table « fils » :

- Prévenir la modification ou la suppression d'une clé primaire (ou candidate) de la table « père ». Cette alternative est celle par défaut. Soit vous n'ajoutez pas d'option à la clause REFERENCES – dans notre exemple, toutes les clés étrangères sont ainsi composées –, soit vous utilisez NO ACTION pour les directives ON DELETE et ON UPDATE. La suppression d'un avion n'est donc pas possible si ce dernier est référencé dans un affrètement.
- Propager la suppression des enregistrements « fils » associés à

l'enregistrement « père » supprimé. Ce mécanisme est réalisé par la directive ON DELETE CASCADE. Dans notre exemple, nous pourrions ainsi décider de supprimer tous les affrètements dès qu'on retire un avion.

- Étendre la modification de la clé primaire de l'enregistrement « père » aux enregistrements « fils » associés. Ce mécanisme est réalisé par la directive ON UPDATE CASCADE. Dans notre exemple, nous pourrions ainsi décider de mettre à jour tous les affrètements dès qu'on modifie l'immatriculation d'un avion.
- Propager l'affectation de la valeur nulle aux clés étrangères des enregistrements « fils » associés à l'enregistrement « père » supprimé ou modifié. Ce mécanisme est réalisé par la directive ON DELETE SET NULL (ou ON UPDATE SET NULL en cas de modification de la clé primaire du « père »). Dans ces deux cas, il ne faut pas poser de contrainte NOT NULL sur la clé étrangère. Dans notre exemple, nous pourrions ainsi décider de mettre NULL dans la colonne compa de la table Pilote pour chaque pilote d'une compagnie supprimée. Nous ne pourrions pas appliquer ce mécanisme à la table Affreter qui dispose de contraintes NOT NULL sur ses clés étrangères (car composant la clé primaire).

RESTRICT est une directive de la norme SQL qui n'est pas encore mise en œuvre sous MySQL. Elle concerne les SGBD compatibles avec les contraintes différées. Pour l'heure NO ACTION (qui en principe diffère les contraintes) et RESTRICT (qui ne diffère pas les contraintes) jouent le même rôle.

Les options DELETE CASCADE et DELETE SET NULL sont disponibles depuis la version 3.23.50, celles relatives à ON UPDATE sont disponibles depuis la version 4.0.8.

Le tableau suivant décrit quelques alternatives de cohérence à notre base de données exemple entre les tables Avion et Affreter puis Pilote et Compagnie.

Tableau 2-20 Alternatives de cohérence du « père » vers les « fils »

Alternatives	Syntaxe / Message d'erreur

Prévenir la modification	`--dans Affreter`
	`CONSTRAINT fk_Aff_na_Avion`
	`FOREIGN KEY(immat) REFERENCES Avion(immat)`
	`ON DELETE NO ACTION ON UPDATE NO ACTION`

de l'immatriculation d'un avion ou la suppression d'un avion.	`mysql> DELETE FROM Avion; ERROR 1451 (23000): Cannot delete or update a parent row: a foreign key constraint fails (`bdsoutou/affreter`, CONSTRAINT `fk_Aff_na_Avion` FOREIGN KEY (`immat`) REFERENCES `avion` (`immat`) ON DELETE NO ACTION ON UPDATE NO ACTION)`

Propager la suppression d'un avion.	`--dans Affreter`
	`CONSTRAINT fk_Aff_na_Avion`
	`FOREIGN KEY(immat) REFERENCES Avion(immat)`
	`ON DELETE CASCADE`

Propager la modification de l'immatriculation d'un avion.	`--dans Affreter`
	`CONSTRAINT fk_Aff_na_Avion`
	`FOREIGN KEY(immat) REFERENCES Avion(immat)`
	`ON UPDATE CASCADE`

Propager la modification du code de la compagnie dans les tables `Pilote`, `Avion` et `Affreter`. Affecter la valeur nulle dans la table `Pilote` suite à la suppression d'une compagnie, tout en préservant l'intégrité avec la table `Avion`.	`--dans Affreter`
	`CONSTRAINT fk_Aff_comp_Compag`
	`FOREIGN KEY(compAff) REFERENCES Compagnie(comp)`
	`ON UPDATE CASCADE`
	`--dans Avion`
	`CONSTRAINT fk_Avion_comp_Compag`
	`FOREIGN KEY(proprio) REFERENCES Compagnie(comp)`
	`ON UPDATE CASCADE`
	`--dans Pilote`
	`CONSTRAINT fk_Pil_compa_Comp`
	`FOREIGN KEY (compa) REFERENCES Compagnie(comp)`
	`ON DELETE SET NULL ON UPDATE CASCADE`

Pour l'heure, aucun déclencheur ne peut être activé suite à la modification d'une colonne induite d'une action de répercussion (`CASCADE` ou `SET NULL`).

MySQL ne permet pas encore de propager une valeur par défaut (*set default*) comme la norme SQL le prévoit.

En résumé

Le tableau suivant résume les conditions requises pour modifier l'état de la base de données en respectant l'intégrité référentielle :

Tableau 2-21 Instructions SQL sur les clés

Instructions	Table « père »	Table « fils »

INSERT	Correcte si la clé primaire (ou candidate) est unique.	Correcte si la clé étrangère est référencée dans la table « père » ou est nulle (partiellement ou en totalité).
UPDATE	Correcte si l'instruction ne laisse pas d'enregistrements dans la table « fils » ayant une clé étrangère non référencée.	Correcte si la nouvelle clé étrangère référence un enregistrement « père » existant.
DELETE	Correcte si aucun enregistrement de la table « fils » ne référence le ou les enregistrements détruits.	Correcte sans condition.
DELETE *Cascade*	Correcte sans condition.	Sans objet.
DELETE SET NULL	Correcte sans condition.	Sans objet.
UPDATE *Cascade*	Correcte sans condition.	Sans objet.
UPDATE SET NULL	Correcte s'il n'y a pas de NOT NULL dans la table « fils ».	Sans objet.

Insertions à partir d'un fichier

L'importation de données (au format texte) dans une table peut se programmer par la commande LOAD DATA INFILE. Un tel mécanisme lit un fichier dans un répertoire du serveur et insère tout ou partie des informations dans une table. Vous devez avoir reçu le privilège global FILE de l'administrateur (autorisation d'accès aux fichiers du système d'exploitation) pour pouvoir exécuter cette commande.

La syntaxe simplifiée de cette directive est la suivante :

```
LOAD DATA INFILE 'nomEtCheminFichier.txt'
    [REPLACE | IGNORE] INTO TABLE nomTable
    [FIELDS [TERMINATED BY 'string']
            [[OPTIONALLY] ENCLOSED BY 'char']
            [ESCAPED BY 'char' ] ]
    [LINES [STARTING BY 'string'] [TERMINATED BY 'string'] ]
    [IGNORE number LINES];
    [col_ou_variable1,...)];
```

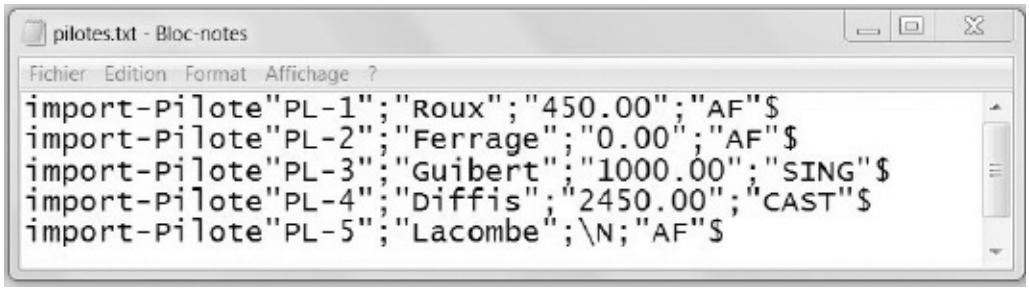
- REPLACE : option à utiliser pour remplacer systématiquement les anciens

enregistrements par les nouveaux (valeur de clé primaire ou d'index unique).

- `IGNORE` : fait en sorte de ne pas insérer des enregistrements de clé primaire ou d'index unique déjà présents dans la table.
- *`col_ou_variable`* : liste éventuelle de colonnes ou de variables.
- `FIELDS` décrit comment sont formatées dans le fichier les valeur à insérer dans la table. En l'absence de cette clause, `TERMINATED BY` vaut `'\t'`, `ENCLOSED BY` vaut `''` et `ESCAPED BY` vaut `'\\'`.
  - `FIELDS TERMINATED BY` décrit le caractère qui sépare deux valeurs de colonnes.
  - `FIELDS ENCLOSED BY` permet de contrôler le caractère qui encadrera chaque valeur de colonne.
  - `FIELDS ESCAPED BY` permet de contrôler les caractères spéciaux.
- `LINES` décrit comment seront écrites dans le fichier les lignes extraites de(s) table(s). En l'absence de cette clause, `TERMINATED BY` vaut `'\n'` et `STARTING BY` vaut `''`.
- `IGNORE` permet de ne pas importer les *nb* premières lignes du fichier (contenant des éventuelles déclarations).

À titre d'exemple, supposons que le contenu du fichier « `pilotes.txt` » (situé dans le répertoire « `C:\Donnees\dev` ») soit le suivant :

Figure 2-12 *Importation de données*



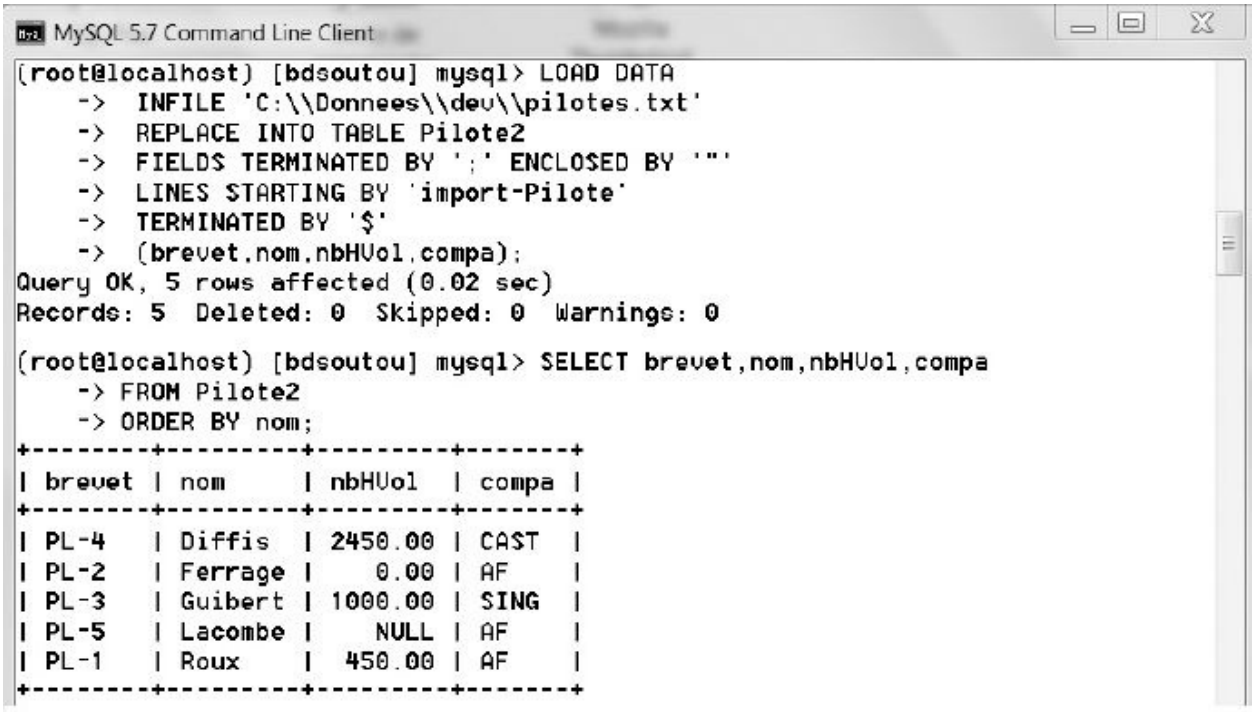
Décidons d'importer ces données dans la table `Pilote2` que vous aurez préalablement créée et dont la structure est la suivante : `brevet VARCHAR(6) PRIMARY KEY, nom VARCHAR(16), nbHVol DECIMAL(7,2), compa VARCHAR(4)`. Notez l'utilisation du double *back-slash* pour désigner une arborescence Windows. Le caractère `NULL` de la colonne `nbHVol` du dernier importé est désigné

par le caractère `\N`. Le caractère `\n` représente le retour à la ligne (mais peut être aussi une source de problèmes selon les versions utilisées).

```
LOAD DATA INFILE 'C:\\Donnees\\dev\\pilotes.txt'
     REPLACE INTO TABLE Pilote2
        FIELDS TERMINATED BY ';' ENCLOSED BY '"'
        LINES STARTING BY 'import-Pilote' TERMINATED BY '$\n';
```

Une fois la table créée, il est possible de l'interroger :

Figure 2-13 *Importation et résultat*



Exercices

Les objectifs de ces exercices sont :

- d'insérer des données dans les tables du schéma *Parc Informatique* ;
- de créer une séquence et d'insérer des données en utilisant une séquence ;
- de modifier des données.

Exercice 2.1

Insertion de données

Écrire puis exécuter le script SQL (que vous appellerez `insParc.sql`) afin d'insérer les données dans les tables suivantes :

Tableau 2-22 Données des tables

Table	Données
Segment	**INDIP** NOMSEGMENT ETAGE 130.120.80 Brin RDC 130.120.81 Brin 1er étage 130.120.82 Brin 2e étage
Salle	**NSALLE** NOMSALLE NBPOSTE INDIP s01 Salle 1 3 130.120.80 s02 Salle 2 2 130.120.80 s03 Salle 3 2 130.120.80 s11 Salle 11 2 130.120.81 s12 Salle 12 1 130.120.81 s21 Salle 21 2 130.120.82 s22 Salle 22 0 130.120.83 s23 Salle 23 0 130.120.83
Poste	**NPOSTE** NOMPOSTE INDIP AD TYPEPOSTE NSALLE p1 Poste 1 130.120.80 01 TX s01 p2 Poste 2 130.120.80 02 UNIX s01 p3 Poste 3 130.120.80 03 TX s01 p4 Poste 4 130.120.80 04 PCWS s02 p5 Poste 5 130.120.80 05 PCWS s02 p6 Poste 6 130.120.80 06 UNIX s03 p7 Poste 7 130.120.80 07 TX s03 p8 Poste 8 130.120.81 01 UNIX s11 p9 Poste 9 130.120.81 02 TX s11 p10 Poste 10 130.120.81 03 UNIX s12 p11 Poste 11 130.120.82 01 PCNT s21 p12 Poste 12 130.120.82 02 PCWS s21
Logiciel	**NLOG** NOMLOG DATEACH VERSION TYPELOG PRIX log1 Oracle 6 1995-05-13 6.2 UNIX 3000 log2 Oracle 8 1999-09-15 8i UNIX 5600 log3 SQL Server 1998-04-12 7 PCNT 2700 log4 Front Page 1997-06-03 5 PCWS 500 log5 WinDev 1997-05-12 5 PCWS 750 log6 SQL*Net 2.0 UNIX 500 log7 I. I. S. 2002-04-12 2 PCNT 810 log8 DreamWeaver 2003-09-21 2.0 BeOS 1400
Types	**TYPELP** NOMTYPE TX Terminal X-Window UNIX Système Unix PCNT PC Windows NT PCWS PC Windows NC Network Computer

Exercice 2.2

Gestion d'une séquence

Dans ce même script, gérer la séquence associée à la colonne `numIns` commençant à la valeur 1 de manière à insérer les enregistrements suivants :

Tableau 2-23 Données de la table `Installer`

Table	Données
Installer	NPOSTE NLOG **NUMINS** DATEINS DELAI p2 log1 1 2003-05-15 p2 log2 2 2003-09-17 p4 log5 3 p6 log6 4 2003-05-20 p6 log1 5 2003-05-20 p8 log2 6 2003-05-19 p8 log6 7 2003-05-20 p11 log3 8 2003-04-20 p12 log4 9 2003-04-20 p11 log7 10 2003-04-20 p7 log7 11 2002-04-01

Exercice 2.3

Modification de données

Écrire le script `modification.sql` qui permet de modifier (avec `UPDATE`) la colonne `etage` (pour l'instant nulle) de la table `Segment`, afin d'affecter un numéro d'étage correct (0 pour le segment 130.120.80, 1 pour le segment 130.120.81, 2 pour le segment 130.120.82).

Diminuer de 10 % le prix des logiciels de type 'PCNT'.

Vérifier :

```
SELECT indIP, nomSegment, etage FROM Segment;
SELECT nLog, typeLog, prix FROM Logiciel;
```

Chapitre 3

Évolution d'un schéma

L'évolution d'un schéma est un aspect très important à prendre en compte, car il répond aux besoins de maintenance des applicatifs qui utilisent la base de données. Nous verrons qu'il est possible de modifier une base de données d'un point de vue structurel (colonnes et index) mais aussi comportemental (contraintes).

L'instruction principalement utilisée est ALTER TABLE (commande du LDD) qui permet d'ajouter, de renommer, de modifier et de supprimer des colonnes d'une table. Elle permet aussi d'ajouter et de supprimer des contraintes. Avant de détailler ces mécanismes, étudions la commande qui permet de renommer une table.

Renommer une table [RENAME]

L'instruction RENAME renomme une ou plusieurs tables ou vues. Il faut posséder le privilège ALTER et DROP sur la table d'origine, et CREATE sur la base.

```
RENAME [nomBase.]ancienNomTable TO [nomBase.] nouveauNomTable
       [,[nomBase.] ancienNom2 TO [nomBase.]nouveauNom2]];
```

Les contraintes d'intégrité, index et prérogatives associés à l'ancienne table sont automatiquement transférés sur la nouvelle. En revanche, les vues et procédures cataloguées sont invalidées et doivent être recréées.

Il est aussi possible d'utiliser l'option RENAME TO de l'instruction ALTER TABLE pour renommer une table existante. Le tableau suivant décrit comment renommer la table Pilote sans perturber l'intégrité référentielle :

Tableau 3-1 Renommer une table

Commande RENAME	Commande ALTER TABLE
`RENAME Pilote TO Navigant;`	`ALTER TABLE Pilote RENAME TO Navigant;`

Modifications structurelles [ALTER TABLE]

Considérons la table suivante que nous allons faire évoluer :

Figure 3-1 *Table à modifier*

```
CREATE TABLE Pilote
   (brevet VARCHAR(4), nom VARCHAR(20));

INSERT INTO Pilote
   VALUES ('PL-1', 'Dubois');
```

Pilote

brevet	nom
PL-1	Dubois

Ajout de colonnes

La directive ADD de l'instruction ALTER TABLE permet d'ajouter une nouvelle colonne à une table. Cette colonne est initialisée à NULL pour tous les enregistrements (à moins de spécifier une contrainte DEFAULT, auquel cas tous les enregistrements de la table sont mis à jour avec une valeur non nulle).

Il est possible d'ajouter une colonne en ligne NOT NULL seulement si la table est vide ou si une contrainte DEFAULT est définie sur la nouvelle colonne (dans le cas inverse, il faudra utiliser MODIFY à la place de ADD).

Le script suivant ajoute trois colonnes à la table Pilote. La première instruction insère la colonne nbHVol en l'initialisant à NULL pour tous les pilotes (ici il n'en existe qu'une seule). La deuxième commande ajoute deux colonnes initialisées à une valeur non nulle. La colonne ville ne sera jamais nulle.

```
ALTER TABLE Pilote ADD (nbHVol DECIMAL(7,2));
ALTER TABLE Pilote
      ADD (compa VARCHAR(4)  DEFAULT 'AF',
           ville VARCHAR(30) DEFAULT 'Paris' NOT NULL);
```

La table est désormais la suivante :

Figure 3-2 *Table après l'ajout de colonnes*

Pilote

brevet	nom	nbHVol	compa	ville
PL-1	Dubois		AF	Paris

Renommer des colonnes

Il faut utiliser l'option CHANGE de l'instruction ALTER TABLE pour renommer une colonne existante. Le nouveau nom de colonne ne doit pas être déjà utilisé dans la table. Le type (avec une éventuelle contrainte) doit être reprécisé. La position de la colonne peut aussi être modifiée en même temps. La syntaxe générale de cette option est la suivante :

```
ALTER TABLE [nomBase].nomTable CHANGE [COLUMN] ancienNom
      nouveauNom typeMySQL [NOT NULL | NULL] [DEFAULT valeur]
          [AUTO_INCREMENT] [UNIQUE [KEY] | [PRIMARY] KEY]
                [COMMENT 'chaine'] [REFERENCES ...]
       [FIRST|AFTER nomColonne];
```

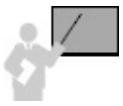
L'instruction suivante permet de renommer la colonne ville en adresse en la positionnant avant la colonne compa :

```
ALTER TABLE Pilote CHANGE ville adresse VARCHAR(30) AFTER nbHVol;
```

Modifier le type des colonnes

L'option MODIFY de l'instruction ALTER TABLE modifie le type d'une colonne existante sans pour autant la renommer. La syntaxe générale de cette instruction est la suivante, les options sont les mêmes que pour CHANGE :

```
ALTER TABLE [nomBase].nomTable MODIFY [COLUMN] nomColonneAmodifier
     typeMySQL [NOT NULL | NULL] [DEFAULT valeur]
          [AUTO_INCREMENT] [UNIQUE [KEY] | [PRIMARY] KEY]
                [COMMENT 'chaine'] [REFERENCES ...]
       [FIRST|AFTER nomColonne];
```



Il est possible d'augmenter la taille d'une colonne numérique (largeur ou précision) – ou d'une chaîne de caractères (CHAR et VARCHAR) – ou de la diminuer si toutes les données présentes dans la colonne peuvent s'adapter à la nouvelle taille.

Attention à ne pas réduire les colonnes indexées à une taille inférieure à celle déclarée lors de la création de l'index.

Les contraintes en ligne peuvent être aussi modifiées par cette instruction. Une fois la colonne changée, les nouvelles contraintes s'appliqueront aux mises à jour ultérieures de la table, et les données présentes devront toutes vérifier cette nouvelle contrainte.

Le tableau suivant présente différentes modifications de colonnes :

Tableau 3-2 Modifications de colonnes

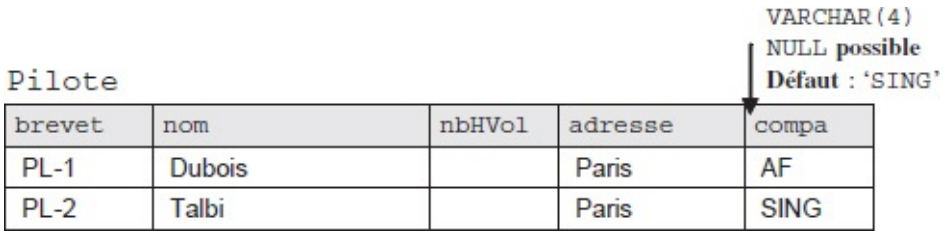
Instructions SQL	Commentaires

`ALTER TABLE Pilote`
`  MODIFY compa VARCHAR(6)`
`  DEFAULT 'SING';`
`INSERT INTO Pilote (brevet, nom)`
`  VALUES ('PL-2', 'Talbi');`
	Augmente la taille de la colonne `compa` et change la contrainte de valeur par défaut puis insère un nouveau pilote.

`ALTER TABLE Pilote` `  MODIFY compa VARCHAR(4) NOT NULL;`	Diminue la colonne et modifie également son type de `VARCHAR` en `CHAR` tout en le déclarant `NOT NULL` (possible car les données contenues dans la colonne ne dépassent pas quatre caractères).
`ALTER TABLE Pilote` `  MODIFY compa VARCHAR(4);`	Rend possible l'insertion de valeur nulle dans la colonne `compa`.

La table est désormais la suivante :

Figure 3-3 *Après modification des colonnes*



Valeurs par défaut

L'option `ALTER COLUMN` de l'instruction `ALTER TABLE` modifie la valeur par défaut

d’une colonne existante. La syntaxe générale de cette instruction est la suivante :

```
ALTER TABLE [nomBase].nomTable ALTER [COLUMN] nomColonneAmodifier
      {SET DEFAULT ‘chaine’ | DROP DEFAULT};
```

Le script suivant définit une valeur par défaut pour la colonne `adresse` puis supprime celle relative à la colonne `compa` :

```
ALTER TABLE Pilote ALTER COLUMN adresse SET DEFAULT 'Blagnac';
ALTER TABLE Pilote ALTER COLUMN compa DROP DEFAULT;
```

Supprimer des colonnes

L’option `DROP` de l’instruction `ALTER TABLE` permet de supprimer une colonne (aussi un index ou une clé que nous étudierons plus loin). La possibilité de supprimer une colonne évite aux administrateurs d’exporter les données, de recréer une nouvelle table, d’importer les données et de recréer les éventuels index et contraintes. Lorsqu’une colonne est supprimée, les index qui l’utilisent sont mis à jour voire éliminés si toutes les colonnes qui le composent sont effacées. La syntaxe des ces options est la suivante :

```
ALTER TABLE [nomBase].nomTable DROP
  { [COLUMN] nomColonne | PRIMARY KEY
    | INDEX nomIndex | FOREIGN KEY nomContrainte }
```

Il n’est pas possible de supprimer avec cette instruction :

- toutes les colonnes d’une table ;
- les colonnes qui sont clés primaires (ou candidates par `UNIQUE`) référencées par des clés étrangères.

La suppression de la colonne `adresse` de la table `Pilote` est programmée par l’instruction suivante :

```
ALTER TABLE Pilote DROP COLUMN adresse;
```

Énumérations

Les énumérations ENUM et SET étudiées au chapitre 2 peuvent évoluer dans le temps. MySQL préserve la cohérence entre les valeurs de l'énumération et les valeurs de la colonne des enregistrements de la table.

Considérons à nouveau l'exemple du diplôme de chaque étudiant. Le tableau suivant décrit l'évolution de la structure de la colonne ENUM.

- Initialement, quatre valeurs non nulles sont permises (ainsi que l'absence de valeur).
- La première modification de la table consiste à ajouter la valeur 'IUP' à la liste et à ne plus permettre d'absence de valeur.
- La deuxième modification de la table consiste à réduire (en préservant la cohérence avec les enregistrements déjà présents dans la table) la liste des diplômes en otant la valeur 'DUT' à la liste. De plus, l'absence de valeur est de nouveau permise.
- La troisième modification de la table tente de modifier la liste mais la cohérence avec les enregistrements déjà présents dans la table ne le permet pas (les valeurs de référence des étudiants 'E1' et 'E2' disparaîtraient alors). Le message d'erreur est le même que celui renvoyé lors de la mise à jour d'un enregistrement incorrect.

Il n'est pas encore possible de récupérer automatiquement les enregistrements qui ne respectent pas la nouvelle définition de l'énumération (ce qu'Oracle réalise avec la clause EXCEPTIONS INTO de l'instruction ALTER TABLE).

Tableau 3-3 Modification d'un ENUM

Évolution de la table	Mises à jour possibles

`CREATE TABLE UnCursus`
`(num CHAR(4), nom CHAR(15), diplome`
`ENUM ('BTS','DUT','Licence','INSA') NULL,`
`CONSTRAINT pk_UnCursus PRIMARY KEY(num));`
	`INSERT INTO UnCursus (num,nom,diplome)`
	`VALUES ('E1', 'Brouard', ('BTS'));`
	`INSERT INTO UnCursus (num,nom,diplome)`
	`VALUES ('E2', 'Difis', 'Licence');`

`/* Ajout de IUP à la liste + pas de NULL */`
 	
`ALTER TABLE UnCursus`
`MODIFY diplome`
	`INSERT INTO UnCursus (num,nom,diplome)`
	`VALUES ('E3','I. IUP', 'IUP');`
	`INSERT INTO UnCursus (num,nom,diplome)`

```
SET ('IUP','BTS','DUT','Licence','INSA') NOT          VALUES ('E4','I. INSA', ('INSA'));
```

```
/* Plus de DUT à la liste mais OK pour les NULL
*/
ALTER TABLE UnCursus
   MODIFY diplome
   SET ('IUP','BTS','Licence','INSA') NULL;
```

```
INSERT INTO UnCursus (num,nom)
   VALUES ('E5','Nullos');
```

```
/* Réduction de la liste mais problème
d'incohérence */
ALTER TABLE UnCursus
   MODIFY diplome SET ('IUP','INSA');
ERROR 1265 (01000): Data truncated for column
'diplome' at row 1
```

L'évolution d'un type SET est analogue dans le principe. La seule différence réside dans le fait que les colonnes des enregistrements peuvent contenir plusieurs valeurs (qui doivent être incluses dans la liste de référence : le SET).

Colonnes virtuelles

Depuis la version 5.7.6, MySQL dispose de la fonctionnalité des colonnes virtuelles. La particularité d'une colonne virtuelle réside dans le fait qu'elle peut ne pas être stockée mais simplement évaluée à la volée (au sein d'une requête, ou d'une instruction de mise à jour). Par analogie, les vues (étudiées au chapitre 5) sont des tables virtuelles.

Au niveau de la création d'une table, la syntaxe à adopter est la suivante :


```
colonne type_SQL [GENERATED ALWAYS] AS (expression)
[VIRTUAL|STORED] [UNIQUE [KEY]]
[[PRIMARY] KEY] [NOT NULL]
[COMMENT texte]
```

- L'expression qui suit la directive AS détermine la valeur scalaire de la colonne.
- VIRTUAL dénote une colonne virtuelle non stockée et non indexable, GENERATED ALWAYS rend le code plus explicite.
- STORED dénote une colonne virtuelle stockée et indexable.

Le script suivant déclare la table `Avion` comportant deux colonnes virtuelles (une non-persistante qui précalcule le nombre d'heures de vol par mois, l'autre persistante qui fournit une seconde clé d'identification).

```
CREATE TABLE Avion
(immat         VARCHAR(6),
 type_avion    VARCHAR(15),
 nbh_vol       DECIMAL(10,2),
 age           DECIMAL(4,1),
 freq_vol_mois DECIMAL(8,2)
               GENERATED ALWAYS AS (nbh_vol/age/12) VIRTUAL,
 nb_pax        SMALLINT,
 type_immat    VARCHAR(22)
               GENERATED ALWAYS AS (CONCAT(type_avion,'-',immat))
                     STORED NOT NULL UNIQUE KEY,
 CONSTRAINT pk_Avion PRIMARY KEY(immat));
```

La description de cette table fait apparaître le type de chaque colonne virtuelle.

```
mysql> DESC Avion;
+--------------+---------------+------+-----+---------+--------------------+
| Field        | Type          | Null | Key | Default | Extra              |
+--------------+---------------+------+-----+---------+--------------------+
| immat        | varchar(6)    | NO   | PRI | NULL    |                    |
| type_avion   | varchar(15)   | YES  |     | NULL    |                    |
| nbh_vol      | decimal(10,2) | YES  |     | NULL    |                    |
| age          | decimal(4,1)  | YES  |     | NULL    |                    |
| freq_vol_mois | decimal(8,2)  | YES  |     | NULL    | VIRTUAL GENERATED  |
| nb_pax       | smallint(6)   | YES  |     | NULL    |                    |
| type_immat   | varchar(22)   | NO   | UNI | NULL    | STORED GENERATED   |
+--------------+---------------+------+-----+---------+--------------------+
```

En ajoutant deux lignes et en extrayant les valeurs de ces colonnes particulières, il vient :

Figure 3-4 *Colonnes virtuelles*

```
mysql> INSERT INTO Avion (immat,type_avion,nbh_vol,age,nb_pax)
    ->        VALUES ('F-WTSS', 'Concorde', 20000, 18, 90);
Query OK, 1 row affected, 1 warning (0.01 sec)

mysql> INSERT INTO Avion (immat,type_avion,nbh_vol,age,nb_pax)
    ->        VALUES ('F-GHTY', 'A380', 450, 0.5, 460);
Query OK, 1 row affected (0.00 sec)

mysql>
mysql> SELECT type_immat,freq_vol_mois FROM Avion;
+-----------------+---------------+
| type_immat      | freq_vol_mois |
+-----------------+---------------+
| A380-F-GHTY     |         75.00 |
| Concorde-F-WTSS |         92.59 |
+-----------------+---------------+
```

L'expression de définition d'une colonne virtuelle ne peut pas faire référence à une autre colonne virtuelle et ne peut être construite qu'avec des colonnes d'une même table.

Modifications comportementales

Nous étudions dans cette section les mécanismes d'ajout, de suppression, d'activation et de désactivation de contraintes.

Faisons évoluer le schéma suivant. Les seules contraintes existantes sont les clés primaires nommées `pk_Compagnie`, pour la table `Compagnie`, et `pk_Avion` pour la table `Avion`.

Figure 3-5 *Schéma à faire évoluer*

Compagnie

comp	nrue	rue	ville	nomComp
AF	10	Gambetta	Paris	Air France
SING	7	Camparols	Singapour	Singapore AL

Affreter

compAff	immat	dateAff	nbPax
AF	F-WTSS	2003-05-13	85
SING	F-GAFU	2003-02-05	155
AF	F-WTSS	2003-05-15	82

Avion

immat	typeAvion	nbHVol	proprio
F-WTSS	Concorde	6570	SING
F-GAFU	A320	3500	AF
F-GLFS	TB-20	2000	SING

Ajout de contraintes

Jusqu'à présent, nous avons créé les contraintes en même temps que les tables. Il est possible de créer des tables sans contraintes (dans ce cas l'ordre de génération n'est pas important et on peut même les élaborer par ordre alphabétique), puis d'ajouter les contraintes. Les outils de conception (*Win'Design*, *Designer* ou *PowerAMC*) adoptent cette démarche lors de la génération automatique de scripts SQL.

La directive `ADD CONSTRAINT` de l'instruction `ALTER TABLE` permet d'ajouter une contrainte à une table. Il est aussi possible d'ajouter un index. La syntaxe

générale est la suivante :

```
ALTER TABLE [nomBase].nomTable ADD
  { INDEX [nomIndex] [typeIndex] (nomColonne1,...)
  | CONSTRAINT nomContrainte typeContrainte }
```

Trois types de contraintes sont possibles :

- `UNIQUE (colonne1 [,colonne2]...)`
- `PRIMARY KEY (colonne1 [,colonne2]...)`
- `FOREIGN KEY (colonne1 [,colonne2]...)`

```
    REFERENCES nomTablePère (col1 [,col2]...)
     [ON DELETE {RESTRICT | CASCADE | SET NULL | NO ACTION}]
     [ON UPDATE {RESTRICT | CASCADE | SET NULL | NO ACTION}]
```

Unicité

Ajoutons la contrainte d'unicité portant sur la colonne du nom de la compagnie. Un index est automatiquement généré sur cette colonne à présent :

```
ALTER TABLE Compagnie ADD CONSTRAINT un_nomC UNIQUE (nomComp);
```

Clé étrangère

Ajoutons la clé étrangère (indexée) à la table `Avion` au niveau de la colonne `proprio` en lui assignant une contrainte `NOT NULL` :

```
ALTER TABLE Avion ADD INDEX (proprio);
ALTER TABLE Avion ADD CONSTRAINT fk_Avion_comp_Compag
                  FOREIGN KEY(proprio) REFERENCES Compagnie(comp);
ALTER TABLE Avion MODIFY proprio CHAR(4) NOT NULL;
```

Clé primaire

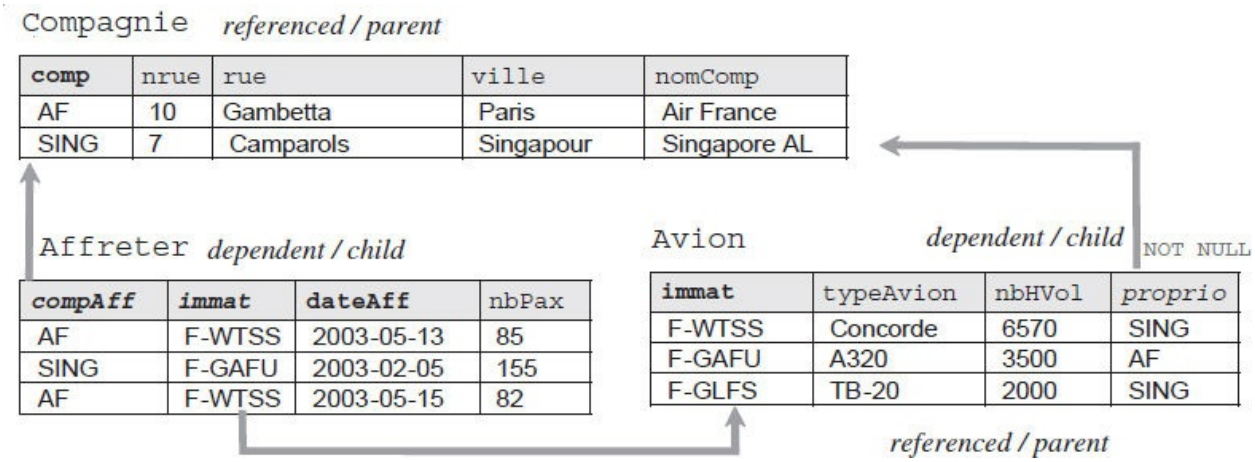
Ajoutons à la table `Affreter`, en une seule instruction, sa clé primaire et deux clés étrangères (une vers la table `Avion` et l'autre vers `Compagnie`) :

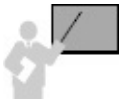
```
ALTER TABLE Affreter ADD (
      CONSTRAINT pk_Affreter        PRIMARY KEY (compAff, immat, dateAff),
      CONSTRAINT fk_Aff_na_Avion    FOREIGN KEY(immat) REFERENCES
                                   Avion(immat),
      CONSTRAINT fk_Aff_comp_Compag FOREIGN KEY(compAff)
                                   REFERENCES Compagnie(comp));
```

Pour que l'ajout ou la modification d'une contrainte soient possibles, il faut que les données présentes dans la table concernée ou référencée respectent la nouvelle contrainte.

Les tables contiennent à présent les contraintes suivantes :

Figure 3-6 *Après ajout de contraintes*





MySQL retourne une erreur lorsqu'il existe au préalable un (des) enregistrement(s) dans une table qui ne respecte(nt) pas la déclaration d'une nouvelle contrainte. En fonction du type de contrainte, les erreurs sont les suivantes :

- unicité : `ERROR 1062 (23000): Duplicate entry '…' for key '…'` ;
- non nullité : `ERROR 1265 (01000): Data truncated for column '…'` ;
- intégrité référentielle : `ERROR 1452 (23000): Cannot add or update a child row: a foreign key constraint fails, CONSTRAINT '…'` ;
- clé primaire : `ERROR 1062 (23000): Duplicate entry …' for key 'PRIMARY'.`

Suppression de contraintes

L'option DROP CONSTRAINT n'existe pas encore pour l'instruction ALTER TABLE permettant de supprimer une contrainte d'un nom donné. Vous devrez donc utiliser une directive de l'instruction ALTER TABLE propre à chaque type de contrainte.

L'absence de contrainte CHECK peut expliquer ce mode de programmation. En conséquence, seul le nom que vous avez donné à vos contraintes de clés étrangères vous sera utile et vous devrez choisir parmi MODIFY, DROP INDEX, DROP FOREIGN KEY et DROP PRIMARY KEY pour supprimer dynamiquement une de vos contraintes.

Contrainte NOT NULL

Il faut utiliser la directive MODIFY de l'instruction ALTER TABLE pour supprimer une contrainte NOT NULL existant sur une colonne. Dans notre exemple, détruisons la contrainte NOT NULL de la clé étrangère proprio dans la table Avion.

```
ALTER TABLE Avion MODIFY proprio CHAR(4) NULL;
```

Contrainte UNIQUE

Il faut utiliser la directive DROP INDEX de l'instruction ALTER TABLE pour supprimer une contrainte d'unicité. Dans notre exemple, effaçons la contrainte portant sur le nom de la compagnie. L'index est également détruit.

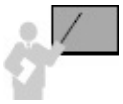
```
ALTER TABLE Compagnie DROP INDEX un_nomC;
```

Clé étrangère

L'option DROP FOREIGN KEY de l'instruction ALTER TABLE permet de supprimer une clé étrangère d'une table. La syntaxe générale est la suivante :

```
ALTER TABLE [nomBase].nomTable DROP FOREIGN KEY nomContrainte;
```

Le nom de la contrainte est celui qui a été déclaré lors de la création de la table, soit lors de sa modification (dans CREATE TABLE ou ALTER TABLE).



Si la contrainte a été définie sans être nommée, son nom est généré par le moteur `InnoDB` et l'instruction `SHOW CREATE TABLE [nomBase].nomTable` permet de le découvrir.

Détruisons la clé étrangère de la colonne `proprio`.

```
ALTER TABLE Avion DROP FOREIGN KEY fk_Avion_comp_Compag;
```

Clé primaire

L'option `DROP PRIMARY KEY` de l'instruction `ALTER TABLE` permet de supprimer une clé primaire. Dans des versions précédentes de MySQL, si aucune clé primaire n'existait, la première contrainte `UNIQUE` disparaissait à la place. Ce n'est plus le cas depuis la version 5.1.

Si la colonne clé primaire à supprimer contient des clés étrangères, il faut d'abord retirer les contraintes de clé étrangère. Si la clé primaire à supprimer est référencée par des clés étrangères d'autres tables, il faut d'abord ôter les contraintes de clé étrangère de ces autres tables.

Ainsi, pour supprimer la clé primaire de la table `Affreter`, il faut d'abord enlever les deux contraintes de clé étrangère concernant des colonnes composant la clé primaire.

```
ALTER TABLE Affreter DROP FOREIGN KEY fk_Aff_na_Avion;
ALTER TABLE Affreter DROP FOREIGN KEY fk_Aff_comp_Compag;


ALTER TABLE Affreter DROP PRIMARY KEY;
```

La figure suivante illustre les contraintes qui restent actives : les clés primaires des tables `Compagnie` et `Avion`, et une contrainte de non nullité.

Figure 3-7 *Après suppression de contraintes*

Compagnie

comp	nrue	rue	ville	nomComp
AF	10	Gambetta	Paris	Air France
SING	7	Camparols	Singapour	Singapore AL

Affreter

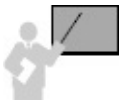
compAff	immat	dateAff	nbPax
AF	F-WTSS	2003-05-13	85
SING	F-GAFU	2003-02-05	155
AF	F-WTSS	2003-05-15	82

Avion

NULL

immat	typeAvion	nbHVol	proprio
F-WTSS	Concorde	6570	SING
F-GAFU	A320	3500	AF
F-GLFS	TB-20	2000	SING

Aucun ordre particulier n'est nécessaire pour supprimer ces trois contraintes, car il n'y a plus de contrainte référentielle active.



Concernant tout schéma, il faut éliminer d'abord les contraintes de clé étrangère des tables « fils » puis « père » puis les contraintes de clé primaire. Il suffit, pour éviter toute incohérence, de détruire les contraintes dans l'ordre inverse d'apparition dans le script de création.

Désactivation des contraintes

La désactivation des contraintes référentielles peut être intéressante pour accélérer des procédures de chargement d'importation et d'exportation massives de données externes. Ce mécanisme améliore aussi les performances de programmes *batchs* qui ne modifient pas des données concernées par l'intégrité référentielle, ou pour lesquelles on vérifie la cohérence de la base à la fin. En effet les index ne sont pas mis à jour pour chaque insertion ou modification, et aucun ordre n'est requis pour insérer ou supprimer des enregistrements.

Syntaxe

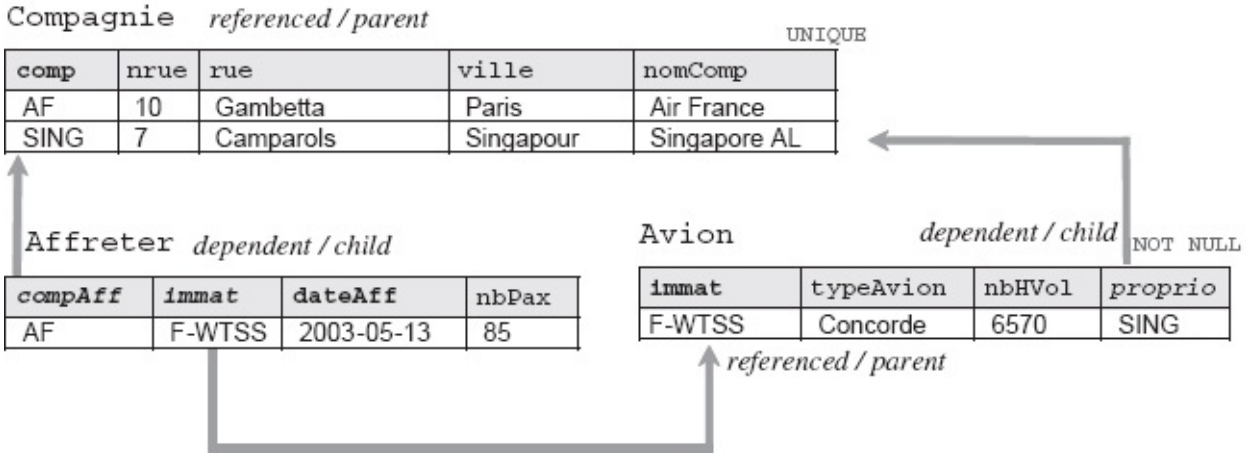
L'instruction `SET FOREIGN_KEY_CHECKS=0` permet de désactiver temporairement (jusqu'à la réactivation) toutes les contraintes référentielles d'une base.

N’essayez pas de désactiver l’intégrité référentielle avec l’option `DISABLE KEYS` de l’instruction `ALTER TABLE`. Cette option concerne les tables `MyISAM` et spécifie seulement de ne pas mettre à jour les index non uniques.

Exemple

En considérant l’exemple suivant, désactivons les contraintes d’intégrité référentielle et tentons d’insérer des enregistrements ne respectant aucune contrainte (d’intégrité référentielle et autres).

Figure 3-8 *Avant la désactivation de contraintes*



On remarque bien que seules les clés étrangères ne sont plus vérifiées. Toute autre contrainte (`UNIQUE`, `PRIMARY KEY` et `NOT NULL`) reste active. L’état de la base est désormais comme suit.

Bien qu’il semble incohérent de réactiver les contraintes sans s’occuper au préalable des valeurs ne respectant pas l’intégrité référentielle (données notées en gras), nous verrons qu’il est possible de le faire.

Réactivation des contraintes

L’instruction `SET FOREIGN_KEY_CHECKS=1` permet de réactiver toutes les contraintes référentielles d’une base.

Tableau 3-4 Insertions après la désactivation de l'intégrité référentielle

Instructions valides	Instructions non valides
mysql> **SET FOREIGN_KEY_CHECKS**=0;	
mysql> INSERT INTO Avion VALUES ('F-GLFS', 'TB-22', 500, **'Toto'**); Query OK, 1 row affected (0.04 sec) mysql> INSERT INTO Avion VALUES ('**Bidon1**', 'TB-21', 1000, 'AF'); Query OK, 1 row affected (0.10 sec) mysql> INSERT INTO Affreter VALUES ('AF', **'Toto'**, '2005-05-13', 0); Query OK, 1 row affected (0.10 sec) mysql> INSERT INTO Affreter VALUES ('**GTI**', 'F-WTSS', '2005-11-07',10); Query OK, 1 row affected (0.02 sec) mysql> INSERT INTO Affreter VALUES ('**GTI**', '**Toto**', '2005-11-07',40); Query OK, 1 row affected (0.03 sec)	mysql> INSERT INTO Compagnie VALUES ('GTR', 1, 'Brassens', 'Blagnac', **'Air France'**); ERROR **1062** (23000): Duplicate entry 'Air France' for key 2 mysql> INSERT INTO Avion VALUES ('Bidon1', 'TB-20', 2000, **NULL**); ERROR **1048** (23000): Column 'proprio' cannot be null mysql> INSERT INTO Avion VALUES ('F-GLFS', 'TB-21', 1000, 'AF'); ERROR **1062** (23000): Duplicate entry 'F-GLFS' for key 1

***Figure 3-9** Après la désactivation des contraintes référentielles*

Compagnie

comp	nrue	rue	ville	nomComp
AF	10	Gambetta	Paris	Air France
SING	7	Camparols	Singapour	Singapore AL

Avion

immat	typeAvion	nbHVol	proprio
F-WTSS	Concorde	6570	SING
F-GLFS	TB-22	500	Toto

Affreter

compAff	immat	dateAff	nbPax
AF	F-WTSS	2003-05-13	82
AF	Toto	2005-05-13	0
GTI	F-WTSS	2005-11-07	10
GTI	Toto	2005-11-07	40

N'essayez pas de désactiver l'intégrité référentielle avec l'option ENABLE KEYS de l'instruction ALTER TABLE. Cette option concerne les tables MyISAM et spécifie de mettre à jour les index non uniques.

Syntaxe

La réactivation totale de l'intégrité référentielle de la base se programme ainsi :

```
mysql> SET FOREIGN_KEY_CHECKS=1;
```

L'intégrité est assurée de nouveau mais ne concerne que les mises à jour à venir (ajouts d'enregistrements, modifications de colonnes et suppressions d'enregistrements). Les éventuelles données présentes dans les tables qui ne vérifient pas l'intégrité sont toujours en base !

Récupération de données erronées

Bien que MySQL ne dispose pas pour l'heure d'un mécanisme de récupération automatique des enregistrements d'une table ne respectant pas les contraintes, il est toutefois possible de programmer des requêtes (étudiées au chapitre 4) permettant l'extraction de ces enregistrements.

Le tableau suivant décrit les deux requêtes à programmer afin d'extraire les enregistrements posant problème. Ici nous extrayons les avions qui référencent une compagnie inexistante, et les affrètements qui référencent une compagnie inexistante ou un avion inexistant. Une fois extraits, il faudra statuer pour chacun d'eux entre une modification de telle ou telle colonne ou bien une suppression. Il apparaît que quatre enregistrements ne respectent pas des contraintes.

Tableau 3-5 Enregistrements posant problème

Table Avion	Table Affreter

`mysql> SELECT immat,proprio FROM Avion`
`  WHERE proprio NOT IN`
`  (SELECT comp FROM Compagnie);`
`+--------+---------+`
`| immat  | proprio |`
`+--------+---------+`
`| F-GLFS | **Toto**  |`
`+--------+---------+`
	`mysql> SELECT compAff,immat,dateAff`
	`  FROM Affreter WHERE compAff NOT IN`
	`  (SELECT comp FROM Compagnie)`
	`  OR immat NOT IN`
	`  (SELECT immat FROM Avion);`
	`+---------+--------+------------+`
	`| compAff | immat  | dateAff  |`
	`+---------+--------+------------+`
	`| **GTI**  | F-WTSS | 2005-11-07 |`
	`| AF  | **Toto**  | 2005-05-13 |`
	`| **GTI**  | **Toto**  | 2005-11-07 |`
	`+---------+--------+------------+`

Désactivez de nouveau l'intégrité référentielle avant de modifier les enregistrements ne vérifiant pas les contraintes, car, sinon, certaines modifications cohérentes ne pourraient avoir lieu à cause de la vérification référentielle sur des valeurs erronées (dans notre exemple si on veut modifier la compagnie 'GTI' du dernier affrètement, l'erreur portera sur la colonne immat).

Dans notre exemple, choisissons :

- D'affecter la compagnie 'AF' aux avions appartenant à des compagnies non référencées dans la table `Avion`. Notez qu'il faut compléter par deux espaces le code compagnie (`CHAR`, inutile s'il avait été `VARCHAR`), et qu'on utilise la même requête que précédemment.

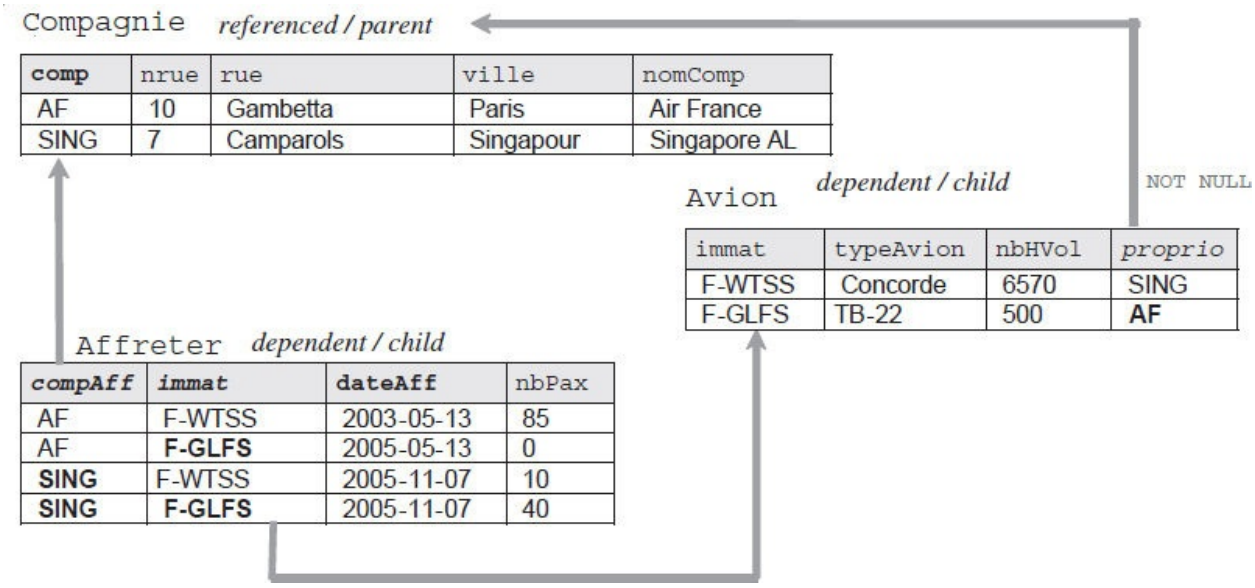
```
SET FOREIGN_KEY_CHECKS=0;
UPDATE Avion SET proprio = 'AF  '
       WHERE proprio NOT IN (SELECT comp FROM Compagnie);
```

- De modifier la compagnie 'GTI' par la compagnie 'SING' dans la table `Affreter` et toute immatriculation d'avion inexistant en 'F-GLFS'.

```
UPDATE Affreter SET compAff ='SING' WHERE compAff = 'GTI ' ;
UPDATE Affreter SET immat='F-GLFS'
       WHERE immat NOT IN (SELECT immat FROM Avion) ;
```

Maintenant, il conviendra de réactiver l'intégrité référentielle. L'état de la base avec les contraintes réactivées est le suivant (les mises à jour sont en gras) :

Figure 3-10 *Tables après modifications et réactivation des contraintes*



Contraintes différées

Les contraintes que nous avons étudiées jusqu'à maintenant sont des contraintes immédiates (*immediate*) qui sont contrôlées après chaque instruction. Une contrainte est dite « différée » (*deferred*) si elle déclenche sa vérification seulement à la fin de la transaction (première instruction `commit` rencontrée). Pour l'heure, MySQL avec InnoDB ne propose pas ce mode de programmation.

Les collations et jeux de caractères

En considérant à nouveau la table des compagnies dont les colonnes sont associées à différentes collations il est possible de modifier tout ou partie de ces associations.

```
CREATE TABLE bdutil.Compagnie
 (comp    CHAR(4), nrue INTEGER(3), rue VARCHAR(20),
  ville   VARCHAR(15) CHARACTER SET latin1 COLLATE latin1_general_ci,
  nomComp VARCHAR(15) CHARACTER SET latin1 COLLATE latin1_general_cs
 ) DEFAULT CHARACTER SET utf8 COLLATE utf8_bin;
```

Le tableau suivant présente le moyen de modifier une collation. Méfiance toutefois, car une conversion peut être défectueuse si des données existantes contiennent des caractères non reconnus dans la nouvelle collation. La commade `SHOW FULL COLUMNS FROM` *`nom_table`* vous renseignera sur la collation de chaque colonne.

Tableau 3-6 Modification d'une collation

Action	Instruction
Modifier la collation par défaut d'une table	`ALTER TABLE bdutil.Compagnie` `  DEFAULT CHARSET utf8 COLLATE utf8_general_ci;`

Ajouter une colonne avec une collation particulière.	`ALTER TABLE bdutil.Compagnie`
	`ADD COLUMN code_comp VARCHAR(5)`
	`  CHARSET utf8 COLLATE utf8_unicode_ci;`

Modifier la collation d'une colonne.	`ALTER TABLE bdutil.Compagnie`
	`MODIFY nomComp VARCHAR(15)`
	`  CHARSET latin1 COLLATE latin1_general_ci;`

Exercices

Les objectifs de ces exercices sont :

- d'ajouter et de modifier des colonnes ;
- d'ajouter des contraintes ;
- de traiter les erreurs.

Exercice 3.1

Ajout de colonnes

Écrire le script `évolution.sql` qui contient les instructions nécessaires pour ajouter les colonnes suivantes (avec `ALTER TABLE`). Le contenu de ces colonnes sera modifié ultérieurement.

Tableau 3-7 Données de la table `Installer`

Table	Nom, type et signification des nouvelles colonnes
`Segment`	`nbSalle TINYINT(2)` : nombre de salles par défaut = 0. `nbPoste TINYINT(2)` : nombre de postes par défaut = 0.
`Logiciel`	`nbInstall TINYINT(2)` : nombre d'installations par défaut = 0.

Poste	nbLog TINYINT(2) : nombre de logiciels installés par défaut = 0.

Vérifier la structure et le contenu de chaque table avec DESCRIBE et SELECT.

Exercice 3.2

Modification de colonnes

Dans ce même script, rajouter les instructions nécessaires pour :

augmenter la taille dans la table Salle de la colonne nomSalle (passer à VARCHAR(30)) ;

diminuer la taille dans la table Segment de la colonne nomSegment à VARCHAR(15) ;

tenter de diminuer la taille dans la table Segment de la colonne nomSegment à VARCHAR(14). Pourquoi la commande n'est-elle pas possible ?

Vérifier la structure et le contenu de chaque table avec DESCRIBE et SELECT.

Exercice 3.3

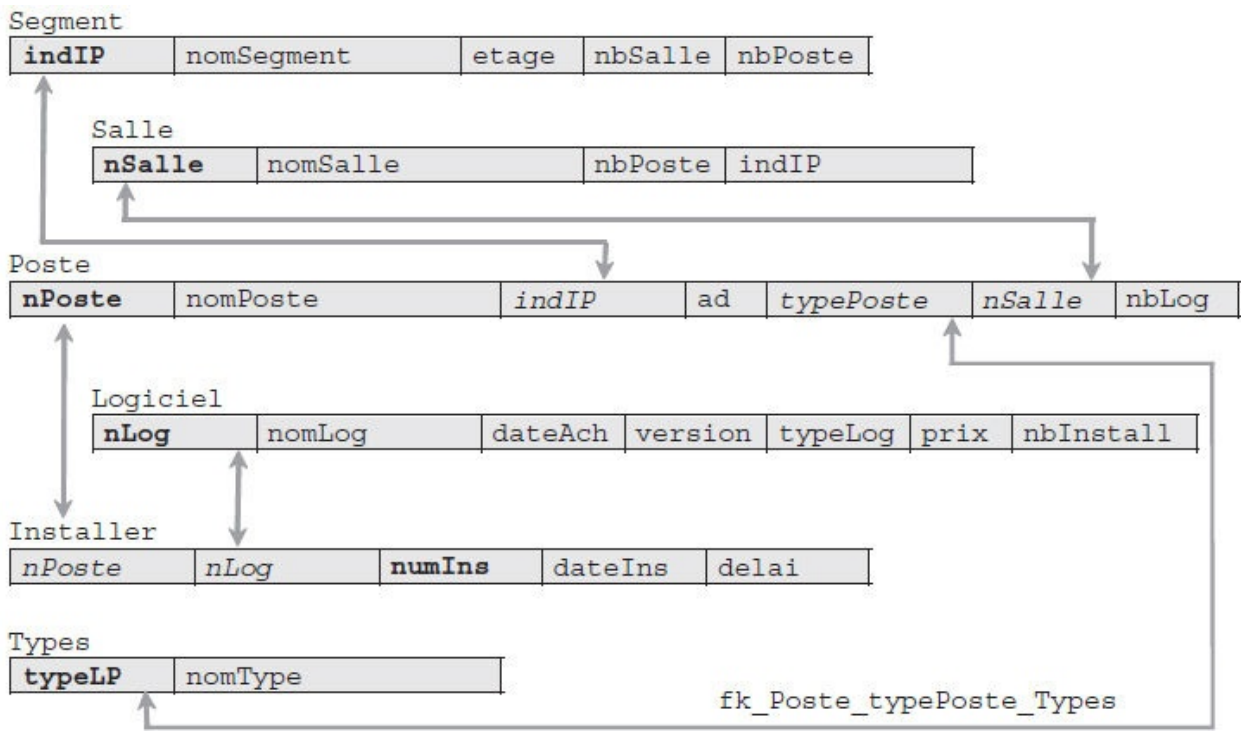
Ajout de contraintes

Ajouter la contrainte afin de s'assurer qu'on ne puisse installer plusieurs fois le même logiciel sur un poste de travail donné.

Ajouter les contraintes de clés étrangères pour assurer l'intégrité référentielle (avec ALTER TABLE… ADD CONSTRAINT…) entre les tables suivantes. Adopter les conventions recommandées dans le chapitre 1 (comme indiqué pour la contrainte entre Poste et Types).

Si l'ajout d'une contrainte référentielle renvoie une erreur, vérifier les enregistrements des tables « pères » et « fils » (notamment au niveau de la casse des chaînes de caractères, 'Tx' est différent de 'TX' par exemple).

Figure 3-11 *Contraintes référentielles à créer*



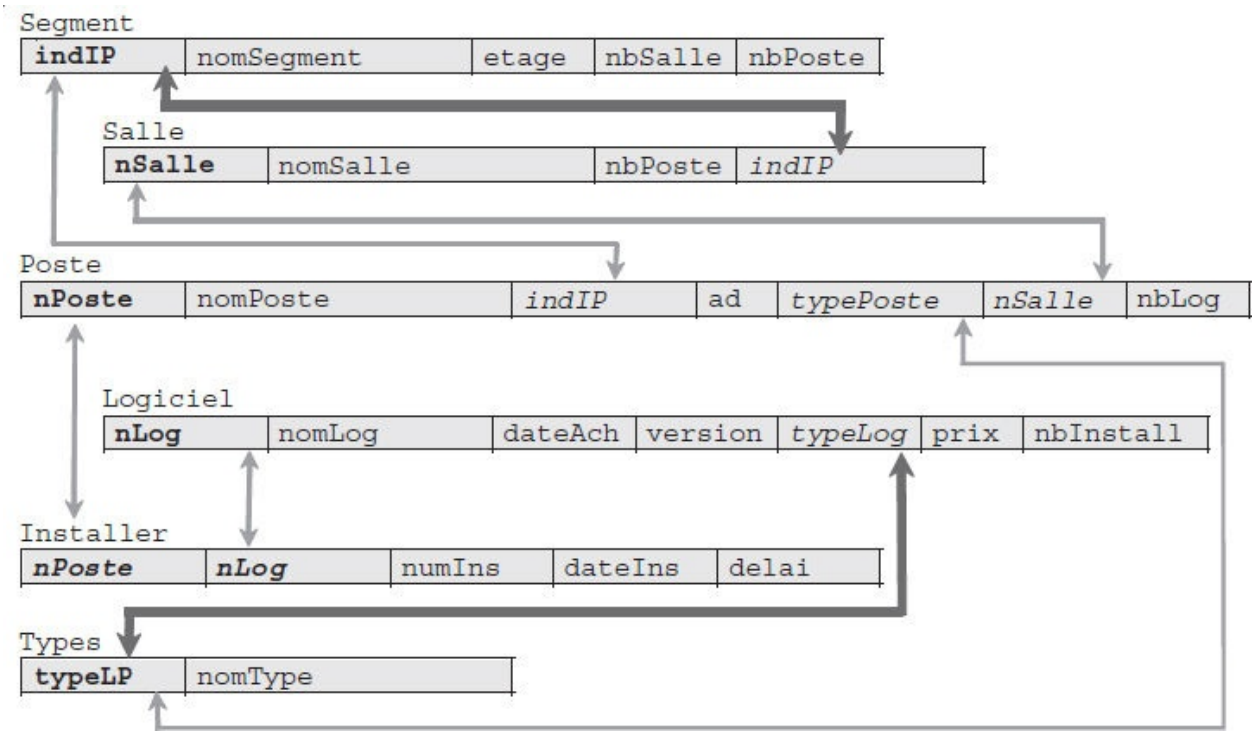
Modifier le script SQL de destruction des tables (`dropParc.sql`) en fonction des nouvelles contraintes. Lancer ce script puis tous ceux écrits jusqu'ici.

Exercice 3.4

Traitements des erreurs

Tentez d'ajouter les contraintes de clés étrangères entre les tables `Salle` et `Segment` et entre `Logiciel` et `Types` (en gras dans le schéma suivant).

Figure 3-12 *Contraintes référentielles à créer*



La mise en place de ces contraintes doit renvoyer une erreur car :

Il existe des salles ('s22' et 's23') ayant un numéro de segment inexistant dans la table `Segment`.

Il existe un logiciel ('log8') dont le type n'est pas référencé dans la table `Types`.

Extraire les enregistrements qui posent problème (numéro des salles pour le premier cas, numéro de logiciel pour le second). Supprimer les enregistrements de la table `Salle` qui posent problème. Ajouter le type de logiciel ('`BeOS`', '`Système Be`') dans la table `Types`.

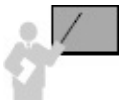
Exécuter à nouveau l'ajout des deux contraintes de clé étrangère. Vérifier que les instructions ne renvoient plus d'erreur et que les deux requêtes d'extraction ne renvoient aucune donnée.

Chapitre 4

Interrogation des données

Ce chapitre traite de l'aspect le plus connu du langage SQL qui concerne l'extraction des données par requêtes (nom donné aux instructions SELECT). Une requête permet de rechercher des données dans une ou plusieurs tables ou vues, à partir de critères simples ou complexes. Les instructions SELECT peuvent être exécutées dans l'interface de commande (voir les exemples de ce chapitre) ou au sein d'un programme SQL (procédure cataloguée), PHP, Java, C, etc.

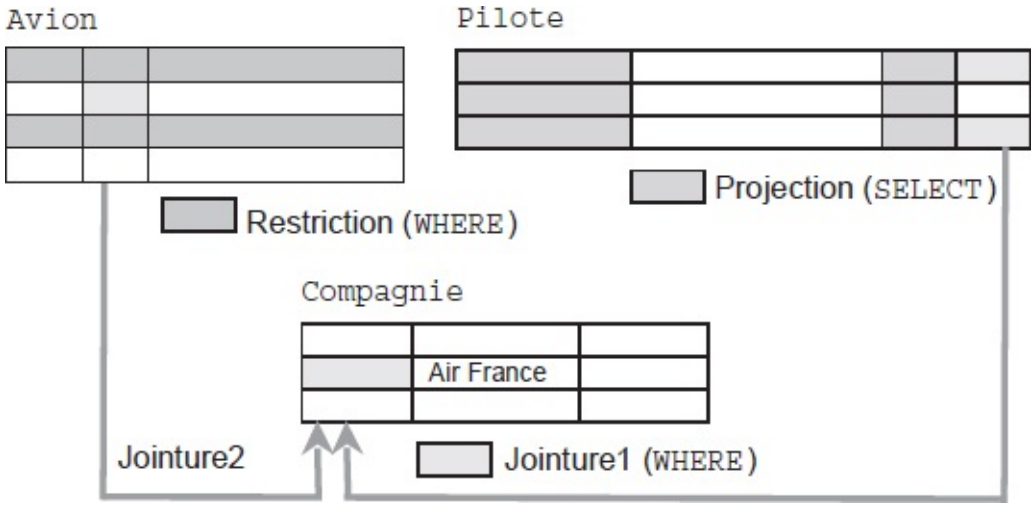
Généralités



L'instruction SELECT est une commande déclarative (elle décrit ce que l'on cherche sans expliquer le moyen d'opérer). À l'inverse, une instruction procédurale (comme un programme) développerait le moyen pour réaliser l'extraction de données (comme le chemin à emprunter entre des tables ou une itération pour parcourir un ensemble d'enregistrements).

La figure suivante schématise les principales fonctionnalités de l'instruction SELECT. Celle-ci est composée d'une directive FROM qui précise la (les) table(s) interrogée(s), et d'une directive WHERE qui contient les critères.

Figure 4-1 *Possibilités de l'instruction SELECT*



- La restriction qui est programmée dans le WHERE de la requête permet de restreindre la recherche à une ou plusieurs lignes. Dans notre exemple une restriction répond à la question : « *Quels sont les avions de type A320 ?* ».
- La projection qui est programmée dans le SELECT de la requête permet d'extraire une ou plusieurs colonnes. Dans notre exemple elle répond à la question : « *Quels sont les numéros de brevet et les nombres d'heures de vol de tous les pilotes ?* ».
- La jointure qui est programmée dans le WHERE de la requête permet d'extraire des données de différentes tables en les reliant deux à deux (le plus souvent à partir de contraintes référentielles). Dans notre exemple la première jointure répond à la question « *Quels sont les numéros de brevet et nombres d'heures de vol des pilotes de la compagnie de nom Air France ?* ». La deuxième jointure répond à la question : « *Quels sont les avions de la compagnie de nom Air France ?* ».

En combinant ces trois fonctionnalités, toute question logique devrait trouver en théorie une réponse par une ou plusieurs requêtes. Les questions trop complexes peuvent être programmées à l'aide des vues (chapitre 5) ou par traitement (programmes mélangeant requêtes et instructions procédurales).

Syntaxe [SELECT]

Pour pouvoir extraire des enregistrements d'une table, il faut que celle-ci soit dans votre base ou que vous ayez reçu le privilège SELECT sur la table.

La syntaxe SQL simplifiée de l'instruction SELECT est la suivante :

```
SELECT [ { DISTINCT | DISTINCTROW } | ALL ] listeColonnes
 FROM nomTable1 [,nomTable2]...
 [ WHERE condition ]
 [ clauseRegroupement ]
 [ HAVING condition ]
 [ clauseOrdonnancement ]
 [ LIMIT [rangDépart,] nbLignes ] ;
```

Nous détaillerons chaque option à l'aide d'exemples au cours de ce chapitre.

Pseudotable

La pseudotable est une table qui n'a pas de nom et qui est utile pour évaluer une expression de la manière suivante : « SELECT *expression*; ». Les résultats fournis seront uniques (si aucune jointure ou opérateur ensembliste ne sont employés dans l'interrogation).

Tableau 4-1 Extraction d'expressions

Besoin	Requête et résultat

Aucun, utilisation probablement la plus superflue.	`mysql> SELECT 'Il reste encore beaucoup de pages?';`
	`+-----------------------------------+`
	`| Il reste encore beaucoup de pages? |`
	`+-----------------------------------+`
	`| Il reste encore beaucoup de pages? |`
	`+-----------------------------------+`

J'ai oublié ma montre !	`mysql> SELECT SYSDATE() "Maintenant : ";`
	`+---------------------+`
	`| Maintenant :  |`
	`+---------------------+`
	`| 2005-11-07 09:15:46 |`
	`+---------------------+`

Pour les matheux qui voudraient retrouver le résultat de 2^{14}, le carré du cosinus de 3π sur 2 et e^{1}.	`mysql> SELECT POWER(2,14),`
	`  POWER(COS(135*3.14159265359/180),2) "Environ"`
	`,EXP(1);`
	`+-------------+-----------------+---------------+`
	`| POWER(2,14) | Environ  | EXP(1)  |`
	`+-------------+-----------------+---------------+`
	`|  16384 | 0.50000000000015 | 2.718281828459 |`
	`+-------------+-----------------+---------------+`

Projection [éléments du SELECT]

Étudions la partie de l'instruction SELECT qui permet de programmer l'opérateur de projection (surligné dans la syntaxe suivante) :

```
SELECT [ { DISTINCT | DISTINCTROW } | ALL ] listeColonnes
 FROM nomTable [aliasTable]
[ clauseOrdonnancement ]
```

```
[ LIMIT [rangDépart,] nbLignes ];
```

- DISTINCT et DISTINCTROW jouent le même rôle, à savoir ne pas inclure les duplicatas.
- ALL prend en compte les duplicatas (option par défaut).
- *listeColonnes* : { * | *expression1* [[AS] *alias1*] [, *expression2* [[AS] *alias2*]...}
  - * : extrait toutes les colonnes de la table.
  - *expression* : nom de colonne, fonction SQL, constante ou calcul.
  - *alias* : renomme l'expression (nom valable pendant la durée de la requête).
- FROM désigne la table (qui porte un alias ou non) à interroger.
- *clauseOrdonnancement* : tri sur une ou plusieurs colonnes ou expressions.
- LIMIT pour limiter le nombre de lignes après résultat.

Interrogeons la table suivante en utilisant chaque option:

***Figure 4-2** Table Pilote*

Pilote

brevet	nom	nbHVol	compa
PL-1	Viel	450	AF
PL-2	Donsez	0	AF
PL-3	Grin	1000	SING
PL-4	Fresnais	2450	CAST
PL-5	Vielle		AF
VARCHAR(6)	VARCHAR(16)	DECIMAL(7,2)	CHAR(4)

Extraction de toutes les colonnes

L'extraction de toutes les colonnes d'une table requiert l'utilisation du symbole « * ».

Tableau 4-2 Utilisation de « * »

Requête SQL	Résultat
mysql> SELECT * FROM Pilote;	+--------+----------+---------+-------+ | brevet | nom | nbHVol | compa | +--------+----------+---------+-------+ | PL-1 | Viel | 450.00 | AF | | PL-2 | Donsez | 0.00 | AF | | PL-3 | Grin | 1000.00 | SING | | PL-4 | Fresnais | 2450.00 | CAST |

```
| PL-5   | Vielle           |    NULL | AF    |
+--------+------------------+---------+-------+
```

Ne codez jamais, à part pour vos tests ou démo, une requête du type « SELECT * … », ceci pour les trois raisons suivantes :

- Moins de données circulent sur le réseau, plus les temps de réponse sont courts. Il est donc préférable d'indiquer dans la liste des colonnes uniquement celles qui sont nécessaires.
- Allégez la charge du transformateur de requêtes en lui évitant de rechercher les informations dans les tables systèmes pour déduire la liste de toutes les colonnes et les privilèges associés.
- Allez-vous interdire implicitement que vos tables évoluent en termes de structure ? Ajouter ou supprimer une colonne risque de rendre le code inopérant à tout endroit où cette instruction se trouvera.

Extraction de certaines colonnes

La liste des colonnes à extraire se trouve dans la clause SELECT.

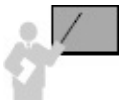
Tableau 4-3 Liste de colonnes

Requête SQL	Résultat

`SELECT compa, brevet FROM Pilote;`	`+-------+--------+`
	`| compa | brevet |`
	`+-------+--------+`
	`| AF    | PL-1   |`
	`| AF    | PL-2   |`
	`| SING  | PL-3   |`
	`| CAST  | PL-4   |`
	`| AF    | PL-5   |`
	`+-------+--------+`

Alias

Les alias permettent de renommer des colonnes à l'affichage ou des tables dans la requête. Les alias de colonnes sont utiles pour les calculs.



L'utilisation de la directive AS est facultative (pour se rendre conforme à SQL2).

Il faut préfixer les colonnes par l'alias de la table lorsqu'il existe.

Tableau 4-4 Alias (colonnes et tables)

Alias de colonnes	Alias de table

```
SELECT compa AS c1,
       nom AS NometPrenom, brevet c3
   FROM Pilote;
+------+-----------------+------+
| c1   | NometPrenom     | c3   |
+------+-----------------+------+
| AF   | Viel            | PL-1 |
| AF   | Donsez          | PL-2 |
| SING | Grin            | PL-3 |
| CAST | Fresnais        | PL-4 |
| AF   | Vielle          | PL-5 |
+------+-----------------+------+
```

```
SELECT aliasPilotes.compa AS c1,
       aliasPilotes.nom
    FROM Pilote aliasPilotes;
+------+-----------------+
| c1   | nom             |
+------+-----------------+
| AF   | Viel            |
| AF   | Donsez          |
| SING | Grin            |
| CAST | Fresnais        |
| AF   | Vielle          |
+------+-----------------+
```

Il semble préférable d'utiliser la directive AS afin qu'il n'y ait pas d'ambiguïté dans l'expression (oubli d'une virgule) `SELECT col1 col2 FROM nomTable` où MySQL interprétera la seconde colonne comme un alias de la première.

Duplicatas

Les directives DISTINCT ou DISTINCTROW éliminent les éventuels duplicatas. Pour la deuxième requête, les écritures « `DISTINCT compa` », « `DISTINCTROW(compa)` » et « `DISTINCTROW compa` » sont équivalentes. La notation entre parenthèses est nécessaire lorsque l'on désire éliminer des duplicatas par paires, triplets, etc.

Tableau 4-5 Gestion des duplicatas

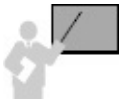
Avec duplicata	Sans duplicata

```
SELECT compa FROM Pilote;
+-------+
| compa |
+-------+
| AF    |
```

```
SELECT DISTINCT(compa) FROM Pilote;
+-------+
| compa |
+-------+
```

```
| AF    |                                    | AF    |
| SING  |                                    | SING  |
| CAST  |                                    | CAST  |
| AF    |                                    +-------+
+-------+
```

Expressions simples



Il est possible d'évaluer et d'afficher simultanément des expressions dans la clause SELECT (types numériques, DATE et DATETIME).

Les opérateurs arithmétiques sont évalués par ordre de priorité (*, /, + et -).

Le résultat d'une expression comportant un NULL est bien souvent un NULL.

Nous avons déjà étudié les opérations sur les dates et intervalles (chapitre 2). Dans l'exemple suivant, l'expression `10*nbHVol+5/2` est calculée en multipliant par 10 le nombre d'heures de vol, puis en ajoutant le résulat de 5 divisé par 2. Dans le second exemple, on convertit le moment actuel (année, mois, jour, heures, minutes et secondes) en un entier.

Tableau 4-6 Expressions numériques

Requête	Résultat

`SELECT brevet, nbHVol, **nbHVol*nbHVol** AS auCarre, **10*nbHVol+5/2** FROM Pilote;`	`+--------+---------+-------------+---------------+`
	`| brevet | nbHVol  | auCarre      | 10*nbHVol+5/2 |`
	`+--------+---------+-------------+---------------+`
	`| PL-1   |  450.00 |  202500.0000 |     4502.5000 |`
	`| PL-2   |    0.00 |       0.0000 |        2.5000 |`
	`| PL-3   | 1000.00 | 1000000.0000 |    10002.5000 |`
	`| PL-4   | 2450.00 | 6002500.0000 |    24502.5000 |`
	`| PL-5   |    NULL |         NULL |          NULL |`
	`+--------+---------+-------------+---------------+`

`SELECT SYSDATE()+0 ;`	`+----------------+`
	`| SYSDATE()+0    |`
	`+----------------+`
	`| 20051107145522 |`
	`+----------------+`

Ordonnancement

Pour trier le résultat d'une requête, il faut spécifier la clause

d’ordonnancement par ORDER BY de la manière suivante :

```
ORDER BY
{ expression1 | position1 | alias1 } [ASC | DESC]
[,{expr2 | position2 | alias2} [ASC | DESC]
```

- *expression* : nom de colonne, fonction SQL, constante, calcul.
- *position* : entier qui désigne l’expression (au lieu de la nommer) dans son ordre d’apparition dans la clause SELECT.
- ASC ou DESC : tri ascendant ou descendant (par défaut ASC).

Dans l’exemple suivant, on remarque que la valeur NULL est considérée comme plus petite que 0.

Tableau 4-7 Ordonnancement

Options par défaut	Ordre décroissant (et NULL)
mysql> SELECT brevet, nom FROM Pilote **ORDER BY** nom; +--------+------------------+ | brevet | nom | +--------+------------------+ | PL-2 | Donsez | | PL-4 | Fresnais | | PL-3 | Grin | | PL-1 | Viel | | PL-5 | Vielle | +--------+------------------+	mysql> SELECT brevet,nbHVol FROM Pilote **ORDER BY** nbHvol **DESC**; +--------+---------+ | brevet | nbHVol | +--------+---------+ | PL-4 | 2450.00 | | PL-3 | 1000.00 | | PL-1 | 450.00 | | PL-2 | 0.00 | | PL-5 | NULL | +--------+---------+

Concaténation

L’opérateur de concaténation se programme à l’aide de la fonction CONCAT qui admet deux chaînes de caractères en paramètre. Cette fonction permet de concaténer différentes expressions (colonnes, calculs, résultats de fonctions SQL ou constantes), sous réserve d’éventuelles conversions (*casting*). La colonne résultante est considérée comme une chaîne de caractères.

L’exemple suivant présente un alias dans l’en-tête de colonne ("Embauche") qui met en forme les résultats. La concaténation concerne deux colonnes et la constante "vole pour".

Tableau 4-8 Concaténation

Requête	Résultat
	+--------+----------

```
SELECT brevet,
 CONCAT(nom,' vole pour ',compa)
 AS "Embauche" FROM Pilote;
```

```
| brevet | Embauche                       |
+--------+--------------------------------+
| PL-1   | Viel vole pour AF              |
| PL-2   | Donsez vole pour AF            |
| PL-3   | Grin vole pour SING            |
| PL-4   | Fresnais vole pour CAST        |
| PL-5   | Vielle vole pour AF            |
+--------+--------------------------------+
```

Insertion multiligne

Nous pouvons maintenant décrire l'insertion multiligne évoquée au chapitre précédent. Dans l'exemple suivant, il s'agit d'insérer tous les pilotes de la table `Pilote` (en considérant le nom, le nombre d'heures de vol et la compagnie) dans la table `NomsetHVoldesPilotes` :

Tableau 4-9 Insertion multiligne

Création et insertion	Requête et résultat

```
CREATE TABLE
 NomsetHVoldesPilotes
 (nom VARCHAR(16),
   nbHVol DECIMAL(7,2),
    compa CHAR(4));
INSERT INTO NomsetHVoldesPilotes
      SELECT nom,nbHVol,compa
           FROM Pilote;
```

```
mysql> SELECT * FROM NomsetHVoldesPilotes;
+------------------+---------+-------+
| nom              | nbHVol  | compa |
+------------------+---------+-------+
| Viel             |  450.00 | AF    |
| Donsez           |    0.00 | AF    |
| Grin             | 1000.00 | SING  |
| Fresnais         | 2450.00 | CAST  |
| Vielle           |    NULL | AF    |
+------------------+---------+-------+
```

Notez que les instructions (`CREATE TABLE` et `INSERT`…) peuvent être programmées en une instruction (option `AS SELECT` de la commande `CREATE TABLE`) :

```
CREATE TABLE NomsetHVoldesPilotes

       AS SELECT nom, nbHVol, compa FROM Pilote;
```

Limitation du nombre de lignes

Pour limiter le nombre de lignes à extraire à partir du résultat d'une requête, il faut spécifier la clause `LIMIT` de la manière suivante :

```
LIMIT [rangDépart,] nbLignes
```

Le premier entier précise le rang de la première ligne sélectionnée (en fonction du tri du résultat). Le second entier indique le nombre maximum de lignes à extraire. La première ligne est considérée comme présente au rang 0.

Ainsi « LIMIT n » équivaut à « LIMIT 0,n ». L'exemple suivant illustre deux utilisations de la clause LIMIT :

Tableau 4-10 Limitation des résultats

Requête	Résultat

Deuxième et troisième (ordre de clé) pilote.
`SELECT * FROM Pilote LIMIT 1,2;`
	`+--------+--------------+---------+-------+`
	`| brevet | nom | nbHVol | compa |`
	`+--------+--------------+---------+-------+`
	`| PL-2 | Donsez | 0.00 | AF |`
	`| PL-3 | Grin | 1000.00 | SING |`
	`+--------+--------------+---------+-------+`

Les deux pilotes les plus expérimentés (par ordre du nombre d'heures de vol).
`SELECT * FROM Pilote ORDER BY nbHvol DESC LIMIT 2;`
	`+--------+-----------------+---------+-------+`
	`| brevet | nom | nbHVol | compa |`
	`+--------+-----------------+---------+-------+`
	`| PL-4 | Fresnais | 2450.00 | CAST |`
	`| PL-3 | Grin | 1000.00 | SING |`
	`+--------+-----------------+---------+-------+`

Restriction [WHERE]

Les éléments de la clause WHERE d'une requête permettent de programmer l'opérateur de restriction. Cette clause limite la recherche aux enregistrements qui respectent une condition simple ou complexe. Cette section s'intéresse à la partie surlignée de l'instruction SELECT suivante :

```
SELECT [ { DISTINCT | DISTINCTROW } | ALL ] listeColonnes
  FROM nomTable [aliasTable]
  [ WHERE condition ] ;
```

- *condition* composée de colonnes, d'expressions, de constantes liées deux à deux entre des opérateurs :
  - de comparaison (>, =, <, >=, <=, <>) ;
  - logiques (NOT, AND ou OR) ;
  - intégrés (BETWEEN, IN, LIKE, IS NULL).

Interrogeons la table suivante en utilisant chaque catégorie d'opérateur :

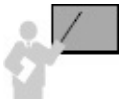
***Figure 4-3** Table Pilote*

Pilote

brevet	nom	nbHVol	prime	compa
PL-1	Viel	450	500	AF
PL-2	Donsez	0		AF
PL-3	Grin	1000	90	SING
PL-4	Fresnais	2450	500	CAST
PL-5	Vielle	400	600	SING
PL-6	Tort		0	CAST

Opérateurs de comparaison

Le tableau suivant décrit des requêtes pour lesquelles la clause `WHERE` contient des opérateurs de comparaison.



Les écritures « `prime=500` » et « (`prime=500`) » sont équivalentes. Les écritures « `prime<>500` », « `NOT` (`prime=500`) » et « `prime<=>500` » sont équivalentes. Les parenthèses sont utiles pour composer des conditions.

Notez l'utilisation du simple guillemet pour comparer des chaînes de caractères.

Tableau 4-11 Égalité, inégalité et comparaison

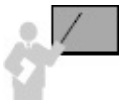
Égalité

```
SELECT brevet, nom AS "Prime 500"
       FROM Pilote WHERE prime = 500 ;
+--------+-----------------+
| brevet | Prime 500       |
+--------+-----------------+
| PL-1   | Viel            |
| PL-4   | Fresnais        |
+--------+-----------------+
SELECT brevet, nom "de Air-France"
       FROM Pilote WHERE compa = 'AF' ;
+--------+--------------+
| brevet | de Air-France |
+--------+--------------+
| PL-1   | Viel         |
| PL-2   | Donsez       |
+--------+--------------+
```

Comparaison et inégalité

```
SELECT brevet, nom, prime
       FROM Pilote WHERE prime <= 400 ;
+--------+---------------+-------+
| brevet | nom           | prime |
+--------+---------------+-------+
| PL-3   | Grin          |    90 |
| PL-6   | Tort          |     0 |
+--------+---------------+-------+
SELECT brevet, nom, prime FROM Pilote
       WHERE prime <> 500 ;
+--------+---------------+-------+
| brevet | nom           | prime |
+--------+---------------+-------+
| PL-3   | Grin          |    90 |
| PL-5   | Vielle        |   600 |
| PL-6   | Tort          |     0 |
+--------+---------------+-------+
```

Opérateurs logiques



- L’ordre de priorité des opérateurs logiques est NOT, AND et OR.
- Les opérateurs de comparaison (>, =, <, >=, <=, <>) sont prioritaires par rapport à NOT.
- Les parenthèses permettent de modifier ces règles de priorité.

La première requête de l’exemple suivant contient une condition composée de trois prédicats qui sont évalués par ordre de priorité (d’abord AND puis OR). La conséquence est l’affichage des pilotes de la compagnie 'SING' avec les pilotes de 'AF' ayant moins de 500 heures de vol.

La deuxième requête force la priorité avec les parenthèses (AND et OR sur le même pied d’égalité). La conséquence est l’affichage des pilotes ayant moins de 500 heures de vol des compagnies 'SING' et 'AF'.

Tableau 4-12 Opérateurs logiques

Requête	Résultat

`SELECT brevet, nom, compa`
` FROM Pilote WHERE`
`(compa = 'SING' OR`
` compa = 'AF' AND nbHVol < 500) ;`
	`+--------+--------------+-------+`
	`| brevet | nom | compa |`
	`+--------+--------------+-------+`
	`| PL-1 | Viel | AF |`
	`| PL-2 | Donsez | AF |`
	`| PL-3 | Grin | SING |`
	`| PL-5 | Vielle | SING |`
	`+--------+--------------+-------+`

`SELECT brevet, nom, compa`
` FROM Pilote WHERE`
`((compa = 'SING' OR compa = 'AF')`
`  AND nbHVol < 500);`
	`+--------+--------------+-------+`
	`| brevet | nom | compa |`
	`+--------+--------------+-------+`
	`| PL-1 | Viel | AF |`
	`| PL-2 | Donsez | AF |`
	`| PL-5 | Vielle | SING |`
	`+--------+--------------+-------+`

Opérateurs intégrés

Les opérateurs intégrés sont BETWEEN, IN, LIKE et IS NULL.

Tableau 4-13 Opérateurs intégrés

Opérateur	Exemple
	`SELECT brevet, nom, nbHVol FROM Pilote` `  WHERE nbHVol BETWEEN 399 AND 1000 ;`

`BETWEEN` *`limiteInf AND limiteSup`* teste l'appartenance à un intervalle de valeurs.

```
+--------+--------------+---------+
| brevet | nom          | nbHVol  |
+--------+--------------+---------+
| PL-1   | Viel         |  450.00 |
| PL-3   | Grin         | 1000.00 |
| PL-5   | Vielle       |  400.00 |
+--------+--------------+---------+
```

`IN` (*`listeValeurs`*) compare une expression avec une liste de valeurs.

```
SELECT brevet, nom, compa FROM Pilote
    WHERE compa IN ('CAST', 'SING') ;
+--------+-----------------+-------+
| brevet | nom             | compa |
+--------+-----------------+-------+
| PL-3   | Grin            | SING  |
| PL-4   | Fresnais        | CAST  |
| PL-5   | Vielle          | SING  |
| PL-6   | Tort            | CAST  |
+--------+-----------------+-------+
```

`LIKE` (*`expression`*) compare de manière générique des chaînes de caractères à une expression.
Le symbole % remplace un ou plusieurs caractères.
Le symbole _ remplace un caractère. Ces symboles peuvent se combiner.
Utilisez de préférence des colonnes `VARCHAR` ou complétez si nécessaire par des blancs jusqu'à la taille maximale pour des `CHAR`.

```
SELECT brevet, nom, compa FROM Pilote
       WHERE compa LIKE ('%A%');
+--------+-----------------+-------+
| brevet | nom             | compa |
+--------+-----------------+-------+
| PL-1   | Viel            | AF    |
| PL-2   | Donsez          | AF    |
| PL-4   | Fresnais        | CAST  |
| PL-6   | Tort            | CAST  |
+--------+-----------------+-------+
SELECT brevet, nom, compa FROM Pilote
       WHERE compa LIKE ('A_');
+--------+--------------+-------+
| brevet | nom          | compa |
+--------+--------------+-------+
| PL-1   | Viel         | AF    |
| PL-2   | Donsez       | AF    |
+--------+--------------+-------+
```

`IS NULL` compare une expression (colonne, calcul, constante) à la valeur `NULL`.
La négation s'écrit soit « *expression* `IS NOT NULL` » soit « `NOT` (*expression* `IS NULL`) ».

```
SELECT nom, prime, nbHVol FROM Pilote
  WHERE prime IS NULL  OR nbHVol IS
NULL ;
+----------------+-------+--------+
| nom            | prime | nbHVol |
+----------------+-------+--------+
| Donsez         |  NULL |   0.00 |
| Tort           |     0 |   NULL |
+----------------+-------+--------+
```

Alias

Il n'est pas permis d'utiliser un alias de colonne dans la clause `WHERE`. Cette recommandation de la norme s'explique par le fait que certaines expressions

pourraient ne pas être déterminées pendant que la condition WHERE est évaluée.

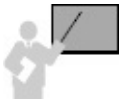
Ainsi, la requête suivante renvoie une erreur alors qu'elle ne contient pourtant pas d'expression litigieuse. Il faudra ici remplacer « c1 » par « aliasDesPilotes.compa ».

```
mysql> SELECT aliasDesPilotes.compa AS c1, aliasDesPilotes.nom
           FROM Pilote aliasDesPilotes WHERE c1 = 'AF';
ERROR 1054 (42S22): Unknown column 'c1' in 'where clause'
```

Fonctions

MySQL propose un grand nombre de fonctions qui s'appliquent dans les clauses SELECT ou WHERE d'une requête. La syntaxe générale d'une fonction est la suivante :

```
nomFonction(colonne1 | expression1 [,colonne2 | expression2 …])
```



- Une fonction monoligne agit sur une ligne à la fois et ramène un résultat par ligne. On distingue quatre familles de fonctions monolignes : caractères, numériques, dates et conversions de types de données. Ces fonctions peuvent se combiner entre elles (exemple : MAX(COS(ABS(n))) désigne le maximum des cosinus de la valeur absolue de la colonne n).
- Une fonction multiligne (fonction d'agrégat) agit sur un ensemble de lignes pour ramener un résultat (voir la section « Regroupements »).

Caractères

Interrogeons la table suivante en utilisant des fonctions pour les caractères :

Figure 4-4 *Table Pilote*

Pilote

brevet	prenom	nom	surnom	compa
PL-1	Gratien	viel	dba	AF
PL-2	Didier	donsez	smith	AF
PL-3	richard	Grin	Faucon	SING
PL-4	placide	Fresnais	cool	CAST
PL-5	Daniel	vielle	jones	SING
PL-6	Francoise	tort	NormaleSup	CAST

La plupart des fonctions pour les caractères acceptent une chaîne de caractères en paramètre de nature CHAR ou VARCHAR. Le tableau suivant décrit les principales fonctions :

Tableau 4-14 Fonctions pour les caractères

Fonction	Objectif	Exemple
`ASCII(c)`	Retourne le caractère ASCII équivalent.	`ASCII ('A')` donne `65`
`CHAR(n)`	Retourne le caractère équivalent dans le jeu de caractères en cours.	`CHAR (161)` donne `í`

`CONCAT(c1,c2)`	Concatène deux chaînes.	`SELECT CONCAT( CONCAT(nom,' vole pour '), compa) "Personnel" FROM Pilote;`
		`+-----------------------------+`
		`| Personnel |`
		`+-----------------------------+`
		`| viel travaille pour AF |`
		`| ... |`

`FIELD(c,c1,c2…)`	Retourne l'index qui correspond à la première égalité entre *c* et *c1*, *c* et *c2*, etc. 0 si aucune égalité n'est trouvée.	`SELECT FIELD('Air', 'air', 'Airbus', 'Air') "Attention à la casse!";`
		`+----------------------+`
		`| Attention à la casse! |`
		`+----------------------+`
		`| 1 |`
		`+----------------------+`

`FIND_IN_SET (c1,c2)`	Recherche dans un ensemble (nombre d'occurrences de *c1* dans *c2* sous-chaînes comprises). *c2* est un ensemble de chaînes séparées par des virgules. *c1* ne doit pas contenir de virgule.	`SELECT FIND_IN_SET('MySQL','Oracle, MySQL,DB2,MySQLAB') "Combien de MySQL";`
		`+------------------+`
		`| Combien de MySQL |`
		`+------------------+`
		`| 2 |`
		`+------------------+`

`INSERT(c1,pos,t,c2)`	Modifie la chaîne *c1* en insérant *t* caractères de la sous-chaîne *c2* à partir de la position *pos*.	`SELECT INSERT('Compxxxie : Airbus ',5, 3, 'agn') "Qui?";`
		`+---------------------+`
		`| Qui? |`
		`+---------------------+`
		`| Compagnie : Airbus |`
		`+---------------------+`

`INSTR(c1,c2)`	Premier indice d'une sous-chaîne *c1* dans une chaîne *c2*. Exemple : indice de 'Air' dans la chaîne.	`SELECT INSTR('Infos-Air : AirBus pour Air-France','Air') "Indice";`
		`+--------+`
		`| Indice |`
		`+--------+`
		`| 7 |`
		`+--------+`

`LOWER(c)`	Tout en minuscules.	`SELECT CONCAT(LOWER(prenom),' ', LOWER(nom)) "Etat civil" FROM Pilote WHERE compa = 'SING';`
		`+---------------+`
		`| Etat civil |`
		`+---------------+`
		`| richard grin |`
		`| daniel vielle |`

	Premier indice d'une sous-	

LOCATE(*c1,c2,pos*)	chaîne *c1* dans une chaîne *c2* à partir de la position *pos*. Exemple : indice de 'Air' dans la chaîne à partir du 9e caractère.	SELECT **LOCATE**('Air','Infos-Air : **A**irBus pour Air-France',9) "Indice après 9"; +---------------+ | Indice après 9 | +---------------+ | 13 | +---------------+
LENGTH(*c*)	Longueur de la chaîne.	SELECT **LENGTH**('Infos-Air : AirBus pour Air-France') "Taille"; +-------+ | Taille | +--------+ | 34 | +--------+
LEFT(*c,n)*	Extrait les *n* premiers caractères à *c* en partant de la gauche.	SELECT **LEFT**('A380 à BlagnacB747B747',14) "Bye les Jumbo"; +---------------+ | Bye les Jumbo | +---------------+ | A380 à Blagnac | +---------------+
LPAD(*c1,n,c2)*	Insertion à gauche de *c2* dans *c1* sur *n* caractères.	SELECT **LPAD**('Rien',20,'-.-') "sur 20"; +---------------------+ | sur 20 | +---------------------+ | -.--.--.--.--Rien | +---------------------+
REPLACE(*c1,c2,c3)*	Recherche les *c2* présentes dans *c1* et les remplace par *c3*.	SELECT **REPLACE**('Matra et Aerospatiale', 'Matra','EADS') "Changement"; +---------------------+ | Changement | +---------------------+ | EADS et Aerospatiale | +---------------------+
REVERSE(*c*)	Retourne la chaîne renversée.	SELECT **REVERSE**('cangalB à 083A') "Miroir"; +---------------+ | Miroir | +---------------+ | A380 à Blagnac | +---------------+
RIGHT(*c,n)*	Extrait les *n* derniers caractères à *c* en partant de la droite.	SELECT **RIGHT**('B747B747A380 à Blagnac',14) "Bye les Jumbo";; +---------------+ | Bye les Jumbo | +---------------+ | A380 à Blagnac | +---------------+

RPAD(*c1,n,c2)*	Insertion à droite de *c2* dans *c1* sur *n* caractères.	SELECT **RPAD**('Rien',19,'-.-') "sur 19"; +--------------------+ | sur 19 | +--------------------+ | Rien-.--.--.--.--.- | +--------------------+
SOUNDEX(*c)*	Extrait la phonétique d'une expression (*in english only !*).	SELECT nom, surnom, compa FROM Pilote WHERE **SOUNDEX**(surnom) IN (**SOUNDEX**('SMYTHE'), **SOUNDEX**('John')); +--------+--------+-------+ | nom | surnom | compa | +--------+--------+-------+ | donsez | **smith** | AF | | vielle | **jone** | SING | +--------+--------+-------+
SUBSTR(*c*,*n*,[*t*])	Extraction de la sous-chaîne *c* commençant à la position *n* sur *t* caractères.	SELECT **SUBSTR**('Air France **à Blagnac** Con!',12,9) "Où ça?"; +-----------+ | Où ça? | +-----------+ | à Blagnac | +-----------+
TRIM(*c1* FROM *c2*)	Enlève les caractères *c1* à la chaîne *c2* (options LEADING et TRAILING pour préciser le sens du découpage). Existent aussi LTRIM et RTRIM qui enlèvent des espaces respectivement au début ou à la fin d'une chaîne.	SELECT **TRIM**('B' FROM 'BA380 à BlagnacBBBBB') "Bye les Jumbo"; +----------------+ | Bye les Jumbo | +----------------+ | A380 à Blagnac | +----------------+
UPPER(*c*)	Tout en majuscules.	SELECT CONCAT(**UPPER**(prenom),' ', **UPPER**(nom)) "Pilotes de CAST" FROM Pilote WHERE compa = 'CAST'; +------------------+ | Pilotes de CAST | +------------------+ | PLACIDE FRESNAIS | | FRANCOISE TORT | +------------------+

Numériques

En plus des opérateurs arithmétiques disponibles dans les langages de programmation (`+`, `-`, `*`, `/` et `DIV` pour la division entière), MySQL supporte un grand nombre de fonctions numériques.

Tableau 4-15 Fonctions numériques

Fonction	Objectif	Exemple
`ABS(`*n*`)`	Valeur absolue de *n*.	
`ACOS(`*n*`)`	Arc cosinus (*n* de -1 à 1), retour exprimé en radians (de 0 à pi).	
`ATAN(`*n*`)`	Arc tangente (∀ *n*), retour exprimé en radians (de -pi/2 à pi/2).	
`CEIL(`*n*`)`	Plus petit entier ≥ à *n*.	`CEIL(15.7)` retourne `16`.
`COS(`*n*`)`	Cosinus de *n* exprimé en radians de 0 à 2 pi .	`COS(60*PI()/180)` retourne `0.5`.
`COT(`*n*`)`	Cotangente de *n* exprimée en radians.	`COT(30*PI()/180)` retourne `1.7320508075689`.
`DEGREES(`*n*`)`	Conversion de radians en degrés.	`DEGREES(PI()/2)` retourne `90`.
`EXP(`*n*`)`	*e* (2.71828183) à la puissance *n*.	
`FLOOR(`*n*`)`	Plus grand entier ≤à *n*.	`FLOOR(15.7)` retourne `15`.
`LN(`*n*`)`	Logarithme népérien de *n*.	
`LOG(`*n*`)(`*m*`,`*n*`)`	Logarithme de *n* dans une base *m*.	
`MOD(`*m*`,`*n*`)`	Reste de la division entière de *m* par *n*.	
`POW(`*m*`,`*n*`)`	*m* puissance *n*.	
`RADIANS(`*n*`)`	Conversion de degrés en radians.	`RADIANS(90)` retourne `1.5707963267949`.

`RAND()`	Flottant aléatoire (à 14 décimales) entre 0 et 1.	Pour obtenir aléatoirement un entier *n* tel que *i*<=*n*<=*j*, utilisez l'expression `FLOOR(i+RAND()*(j-i))`.
`ROUND(`*m*,*n*`)`	Arrondi à une ou plusieurs décimales.	`ROUND(17.567,2)` retourne `17,57`.
`SIGN(`*n*`)`	Retourne le signe d'un nombre (-1, 0 ou 1).	
`SIN(`*n*`)`	Sinus de *n* exprimé en radians de 0 à 2 pi.	`SIN(30*PI()/180)` retourne 0.5.
`SINH(`*n*`)`	Sinus hyperbolique de *n*.	
`SQRT(`*n*`)`	Racine carrée de *n*.	
`TAN(`*n*`)`	Tangente de *n* exprimée en radians de 0 à 2 pi.	
`TRUNCATE(`*n*,*m*`)`	Coupure de *n* à *m* décimales.	`TRUNC(15.79,1)` retourne `15.7`.

Fonctions pour les bits

Les opérateurs suivants sont disponibles jusqu'à 64 bits (`BIGINT`). On peut en utiliser certains en passant des paramètres en base 10, en binaire ou de type chaîne de caractères.

Tableau 4-16 Fonctions pour les bits

Fonction	Objectif	Exemple
OR : `|`	OU bits à bits.	`b'0100' | b'1100'` retourne 12.
AND : `&`	ET bits à bits.	`b'0100' & b'1100'` retourne 4.
XOR : `^`	OU exclusif bits à bits.	`b'0100' ^ b'1100'` retourne 8.
SHL : `<<`	Décalage à gauche de *n* positions.	`3 << 2` retourne 12.
SHR : `>>`	Décalage à droite de *n* positions.	`b'0100' >> 2` retourne 1.
		`3&(~3+1)` retourne 1 (ici

Complément à 1 : ~	Inversion de chaque bit.	on programme le complément à 2).
BIN(*n*)	Chaîne qui représente la valeur binaire de *n*.	`BIN(12)` retourne `'1100'`.
BIT_COUNT (*n*)	Nombre de bits nécessaires pour représenter la valeur *n*.	`BIT_COUNT(9)` retourne 2.
BIT_LENGTH(*c*)	Taille de la chaîne en bits.	`BIT_LENGTH('GTR')` retourne 24 (3 octets).
HEX(*ns*)	Chaîne en hexadécimal représentant *ns* (nombre ou chaîne).	`HEX(254)` retourne 'FE'.
OCT(*n*)	Chaîne en octal représentant *n*.	`OCT(12)` retourne 14.
OCTET_LENGTH(*c*)	Synonyme de `LENGTH()`.	
UNHEX(*c*)	Fonction inverse de HEX.	`UNHEX('53514C')` retourne 'SQL'.

Dates

Le tableau suivant décrit les principales fonctions pour les dates :

Tableau 4-17 Fonctions pour les dates

Fonction	Objectif	Retour
ADDDATE(*date*,*n*)	Ajoute *n* jours à une date (heure).	DATE ou DATETIME
ADDTIME(*date1*,*date2*)	Ajoute les deux dates avec *date1* TIME ou DATETIME, et *date2* TIME.	TIME ou DATETIME
CURDATE(), CURRENT_DATE ou CURRENT_DATE()	Date courante ('YYYY-MM-DD' ou YYYYM-MDD).	INT ou DATE
CURTIME(), CURRENT_TIME ou CURRENT_TIME()	Heure courante ('HH:MM:SS' or HHMMSS).	INT ou DATE
CURRENT_TIMESTAMP,	Date et heure courantes	

CURRENT_TIMESTAMP() ou NOW()	('YYYY-MM-DD HH:MM:SS' ou YYYYMMDDHHMMS).	INT ou DATETIME
DATE(*datet*)	Extrait une date à partir d'une expression de type DATETIME.	DATE
DATEDIFF(*date1*,*date2*)	Nombre entier de jours entre les 2 dates.	INT
DATE_ADD(*date*,INTERVAL *expr type*)	Ajoute un intervalle à une date (heure). *expr* désigne un intervalle. *type* indique comment interpréter le format de l'expression (voir tableau suivant).	DATE ou DATETIME
DATE_FORMAT(*date*,*format*)	Présente la date selon un format (voir tableau suivant).	VARCHAR
DATE_SUB(*date*,INTERVAL *expr type*)	Soustrait un intervalle à une date (heure) Mêmes paramètres que DATE_ADD.	DATE ou DATETIME
DAYNAME(date)	Nom du jour en anglais.	VARCHAR
DAY(date) ou DAYOFMONTH(date)	Numéro du jour dans le mois (0 à 31).	INT
DAYOFYEAR(date)	Numéro du jour dans l'année (0 à 366).	INT
EXTRACT(*type* FROM *date*)	Extrait une partie d'une date selon un type d'intervalle (comme pour DATE_ADD).	INT
FROM_DAYS(*n*)	Retourne une date à partir d'un nombre de jours (le calendrier année 0 débute à *n*=365).	DATE
FROM_UNIXTIME(*nunix*[,*format*])	Retourne une date (heure) à partir d'une estampille Unix (nombre de jours depuis le 1/1/1970). Utilisation possible d'un	INT ou DATETIME

	format.	
HOUR(*time*)	Extrait l'heure d'un temps.	INT
LAST_DAY(*date*)	Dernier jour du mois d'une date (heure).	DATE
LOCALTIME, LOCALTIME(), LOCALTIMESTAMP, LOCALTIMESTAMP()	Synonymes de NOW().	
MAKEDATE(*annee*,*njour*)	Construit une date à partir d'une année et d'un nombre de jours (>0, si *njour*>365, l'année s'incrémente automatiquement).	DATE
MAKETIME(*heure*,*minute*,*seconde*)	Construit une heure.	TIME
MICROSECOND(*date*)	Extrait les microsecondes d'une date-heure.	INT
MINUTE(*time*)	Extrait les minutes d'un temps.	INT
MONTH(*date*), MONTHNAME(*date*)	Retourne respectivement le numéro et le nom du mois d'une date-heure.	INT, VARCHAR
NOW()	Date et heure courantes au format 'YYYY-MM-DD HH:MM:SS' ou YYYYMMDDHHMMSS.	DATETIME ou INT
PERIOD_DIFF(*int1*,*int2*)	Nombre de mois séparant les deux dates au format YYMM or YYYYMM.	INT
SECOND(time)	Extrait les secondes d'un temps.	INT
SEC_TO_TIME(*secondes*)	Construit une heure au format 'HH:MM:SS' ou HHMMSS.	TIME ou INT
STR_TO_DATE(*c*,*format*)	Construit une date (heure) selon un certain format. C'est l'inverse de DATE_FORMAT().	DATE ou DATETIME ou TIME
SUBDATE(*date*,*n*)	Retranche *n* jours à une date (heure).	DATE ou DATETIME

SUBTIME(*date1*,*date2*)	Retranche *date2* (TIME) à *date1* (TIME ou DATETIME).	TIME ou DATETIME
SYSDATE()	Date et heure courantes au format 'YYYY-MM-DD HH:MM:SS' ou YYYYMMDDHHMMSS (différence avec NOW voir chapitre 1).	DATETIME ou INT
TIME(*datetime*)	Extrait le temps d'une date-heure.	TIME
TIMEDIFF(*tdate1*,*tdate2*)	Temps entre 2 temps ou 2 dates ou 2 dates-heure.	TIME
TIMESTAMP(*date*)	Construit une estampille à partir d'une date (heure).	TIMESTAMP
TIMESTAMPADD(*intervalle*,*int*,*date*)	Ajoute à la date (heure) un intervalle (*int*) du type FRAC_SECOND, SECOND, MINUTE, HOUR, DAY, WEEK, MONTH, QUARTER, ou YEAR.	TIMESTAMP
TIMESTAMPDIF(*intervalle*,*int*,*date*)	Retranche à la date (heure) un intervalle du type (idem précédent).	TIMESTAMP
TIME_TO_SEC(*time*)	Retourne le nombre de secondes équivalent au temps.	INT
TO_DAYS(*date*)	Retourne un nombre de jours à partir d'une date ('YYYY-MM-DD' ou YYYYMMDD). Inverse de FROM_DAYS().	INT
UNIX_TIMESTAMP(*date*)	Retourne le nombre de secondes depuis le 1/1/1970 jusqu'à la date (heure) passée en paramètre (ou entier au format YYMMDD YYYYMMDD). Inverse de FROM_UNIXTIME().	INT
UTC_DATE(), UTC_TIME(),	Retournent respectivement la date, l'heure et	DATE, TIME,

`UTC_TIMESTAMP()`	l'estampille au méridien de Greenwich.	`DATETIME`
`WEEKDAY(`*date*`)`	Numéro du jour (0 : *lundi*, 1 : *mardi*, … 6 : *dimanche*) d'une date (heure).	`INT`
`WEEKOFYEAR(`*date*`)`	Numéro de la semaine en cours (1 à 53).	`INT`

Tableau 4-18 Paramètres d'intervalles pour les fonctions `DATE_ADD` et `DATE_SUB`

Paramètre *type*	**Paramètre** *expr*
`MICROSECOND`	*n*
`SECOND`	*n*
`MINUTE`	*n*
`HOUR`	*nn*
`DAY`	*nn*
`WEEK`	*n*
`MONTH`	*nn*
`YEAR`	*nnnn*
`SECOND_MICROSECOND`	`'ss.microsec'`
`MINUTE_MICROSECOND`	`'mi.microsec'`
`MINUTE_SECOND`	`'mi:ssS'`
`HOUR_MICROSECOND`	`'hh.microsec'`
`HOUR_SECOND`	`'hh:mi:ss'`
`HOUR_MINUTE`	`'hh:mi'`
`DAY_MICROSECOND`	`'dd.microsec'`
`DAY_SECOND`	`'dd hh:mi:ss'`
`DAY_MINUTE`	`'dd hh:mi'`
`DAY_HOUR`	`'dd hh'`
`YEAR_MONTH`	`'yyyy-mm'`

Tableau 4-19 Principaux formats pour les fonctions `DATE_FORMAT` et `STR_TO_DATE`

Format	**Description**
`%a`	Nom du jour en anglais abrévié (`Sun`..`Sat`)
`%b`	Nom du mois en anglais abrévié (`Jan`..`Dec`)
`%c`	Mois (`0`..`12`)
`%e`	Jour du mois (`0`..`31`)
`%f`	Microsecondes (`000000`..`999999`)

%H	Heures (00..23)
%i	Minutes (00..59)
%j	Jour de l'année (001..366)
%M	Nom du mois en anglais (January..December)
%S	Secondes (00..59)
%T	*Time* sur 24 heures (hh:mm:ss)
%u	Numéro de semaine (00..53)
%W	Nom du jour en anglais (Sunday..Saturday)
%w	Jour de la semaine (0=Sunday..6=Saturday)
%Y	Année sur 4 positions

Quelques exemples d'utilisation (date du jour : mercredi 9 novembre 2005) sont donnés dans le tableau suivant :

Tableau 4-20 Exemples de fonctions pour les dates

Besoin et fonction	Résultat

Date dans 31 jours.
`SELECT ADDDATE('2005-11-9', 31);`
	`+-------------------------+`
	`| ADDDATE('2005-11-9', 31) |`
	`+-------------------------+`
	`| 2005-12-10 |`
	`+-------------------------+`

1 jour et 1 microseconde après le 9/11/2005, 11 heures, 1 microseconde.
`SELECT ADDTIME('2005-11-09 22:59:59.999999','1 0:0:0.000001') "exemple ADDTIME";`
	`+----------------------------+`
	`| exemple ADDTIME |`
	`+----------------------------+`
	`| 2005-11-10 23:00:00.000000 |`
	`+----------------------------+`

Rendez-vous dans 4 mois. `SELECT DATE_ADD(CURRENT_TIMESTAMP, INTERVAL '4' MONTH) "RDV";`	`+--------------------+`
	`| RDV |`
	`+--------------------+`
	`| 2006-03-09 17:07:33 |`
	`+--------------------+`

Rendez-vous dans 7 jours, 1 heure et 30 minutes.
`SELECT DATE_ADD(CURRENT_TIMESTAMP, INTERVAL '7 01:30:00' DAY_SECOND) "RDV 1sem 1h30";`
	`+--------------------+`
	`| RDV 1sem 1h30 |`
	`+--------------------+`
	`| 2005-11-16 18:53:03 |`
	`+--------------------+`

Aujourd'hui en anglais.
`SELECT DATE_FORMAT(SYSDATE(), '%W %d %M %Y') %Y') "Today in English";`
	`+----------------------------+`
	`| Today in English |`
	`+----------------------------+`
	`| Wednesday 09 November 2005 |`
	`+----------------------------+`

Extraction au format numérique du jour, heures et minutes.	`+------------+`
	`| DAY_MINUTE |`
	`+------------+`

```
SELECT EXTRACT(DAY_MINUTE FROM '2005-11-09 01:02:03') "DAY_MINUTE";
```

```
+------------+
|      90102 |
+------------+
```

Fonctions pour les NULL

En plus des opérateurs `IS NULL` et `IS NOT NULL` qui permettent de filtrer à la demande des `NULL`, il existe des fonctions qui transforment ces valeurs absentes par une valeur choisie. Cette valeur peut être un numérique ou une chaîne de caractères selon la nature de la colonne ou la fonction de conversion utilisée.

La fonction `IFNULL` ressemble à la fonction `NVL` d'Oracle ou `COALESCE` de Microsoft qui serait limitée à un argument. La fonction `NULLIF` est identique à la programmation de : `CASE WHEN expr1=expr2 THEN NULL ELSE expr1 END`.

Tableau 4-21 Fonctions pour les `NULL`

Fonction	Utilisation
IFNULL(*expr1*, *expr2*)	Retourne *expr2* si *expr1* est `NULL` sinon *expr1* est retournée.
NULLIF(*expr1*, *expr2*)	Retourne `NULL` si *expr1* et *expr2* sont identiques, sinon *expr1* est retournée.

En considérant la table présentée à la figure 4-3 de la section « Restriction (`WHERE`) », la première requête remplace une absence d'heures de vol par un quota de 50 et une absence de prime par un libellé. La seconde présente les différences dans le positionnement des paramètres de la fonction `NULLIF`.

Tableau 4-22 Exemples de requêtes avec des fonctions pour les `NULL`

Requête

```
SELECT nom,
       IFNULL(nbHVol,50) AS nbHvol,
       IFNULL(CAST(prime AS CHAR),'aucune')
              AS prime
FROM   Pilote
WHERE  prime IS NULL OR nbHVol IS NULL;
```

Résultat

```
+--------+--------+--------+
| nom    | nbHvol | prime  |
+--------+--------+--------+
| Donsez |   0.00 | aucune |
| Tort   |  50.00 | 0      |
+--------+--------+--------+
```

```
SELECT nom, nbHVol, prime,
NULLIF(nbHVol,prime)
       AS hvol_prime,
NULLIF(prime,nbHVol)
       AS prime_hvol
FROM   Pilote
```

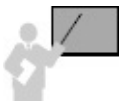
```
+----------+---------+-------+------------+------------+
| nom      | nbHVol  | prime | hvol_prime | prime_hvol |
+----------+---------+-------+------------+------------+
| Donsez   |    0.00 |  NULL |       0.00 |       NULL |
| Fresnais | 2450.00 |   500 |    2450.00 |        500 |
| Grin     | 1000.00 |    90 |    1000.00 |         90 |
| Tort     |    NULL |     0 |       NULL |          0 |
```

```
ORDER BY nom;                  | Viel     |  450.00 |   500 |     450.00 |        500
                               |
                               | Vielle   |  400.00 |   600 |     400.00 |        600
                               |
                               +----------+---------+-------+------------+------------+
```

Conversions

MySQL autorise des conversions de types implicites ou explicites.

Implicites

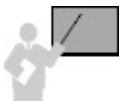


Il est possible d'affecter, dans une expression ou dans une instruction SQL (INSERT, UPDATE…), une donnée de type numérique (ou date-heure) à une donnée de type VARCHAR (ou CHAR). Il en va de même pour l'affectation d'une colonne VARCHAR par une donnée de type date-heure (ou numérique). On parle ainsi de conversions implicites.

Pour preuve, le script suivant ne renvoie aucune erreur :

```
CREATE TABLE Test (c1 DECIMAL(6,3), c2 DATE, c3 VARCHAR(1), c4 CHAR);
INSERT INTO Test VALUES ('548.45', '20060116', 3, 5);
```

Explicites



Une conversion est dite « explicite » quand on utilise une fonction à cet effet. Les fonctions de conversion les plus connues sont CAST et CONVERT (qui respectent la syntaxe de la norme SQL).

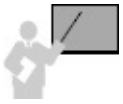
Les fonctions de conversion sont décrites dans le tableau suivant :

Tableau 4-23 Fonctions de conversion

Fonction	Conversion	Exemple
		Pour le premier pilote de

`BINARY(expr)`	L'expression en bits.	notre dernier exemple, le test `BINARY(brevet)` = `BINARY('pl-1')` renverra faux.
`CAST(expression AS typeMySQL)`	L'expression dans le type en paramètre (`BINARY, CHAR, DATE, DATETIME, DECIMAL, SIGNED, TIME, UNSIGNED`).	`CAST(2 AS CHAR)` retourne '2'.
`CONVERT(c, jeucar)`	La chaîne *c* dans le jeu de caractères passé en paramètre.	`CONVERT('Ä Ê Í Ø' USING cp850)` jeu de caractère DOS, retourne `"? Ê Í ?"`.

Comparaisons



MySQL compare deux variables entre elles en suivant les règles suivantes :

- Si l'une des deux valeurs est `NULL`, la comparaison retourne `NULL` (sauf pour l'opérateur <=> qui renvoie vrai si les deux valeurs sont `NULL`).
- Si les deux valeurs sont des chaînes, elles sont comparées en tant que telles.
- Si les deux valeurs sont des numériques, elles sont comparées en tant que telles.
- Les valeurs hexadécimales sont traitées comme des chaînes de bits si elles ne sont pas comparées à des numériques.
- Si l'une des valeurs est `TIMESTAMP` ou `DATETIME` et si l'autre est une constante, cette dernière est convertie en `TIMESTAMP`.
- Dans les autres cas, les valeurs sont comparées comme des numériques (flottants).

Énumérations

Nous avons vu au chapitre 2 comment manipuler les deux types d'énumérations que MySQL propose (`ENUM` et `SET`). Étudions à présent quelques fonctions relatives à ces types.

Type ENUM

Pour tout enregistrement, chaque valeur d'une colonne de type ENUM est associée à un indice (qui commençe à 1). Ainsi, il est possible de retrouver la position d'une valeur au sein de la liste de référence composant l'énumération comme l'illustre l'exemple suivant. L'expression `diplome+0` permet de convertir la colonne en entier. Il apparaît que l'indice d'une expression NULL est NULL.

Tableau 4-24 Extraction des indices d'une énumération ENUM

Table et données	Extraction

`CREATE TABLE UnCursus`
`(num CHAR(4), nom CHAR(15),`
`diplome ENUM`
`('BTS','DUT','INSA','Licence',''),`
`CONSTRAINT pk_Cursus PRIMARY`
`KEY(num));`
 	
`INSERT INTO UnCursus`
`VALUES ('E1','Caboche','DUT');`
`INSERT INTO UnCursus`
`VALUES ('E2','Brouard','BTS');`
`INSERT INTO UnCursus`
`VALUES`
`('E3','Degrelle','Licence');`
`INSERT INTO UnCursus`
`VALUES ('E4','Bruchez','');`
`INSERT INTO UnCursus`
`VALUES ('E5','Zurfluh',NULL);`
	`mysql> SELECT num, nom, diplome, diplome+0`
	`FROM UnCursus;`
	
	`+-----+-------------+---------+-----------+`
	`| num | nom | diplome | diplome+0 |`
	`+-----+-------------+---------+-----------+`
	`| E1 | Caboche | DUT | 2 |`
	`| E2 | Brouard | BTS | 1 |`
	`| E3 | Degrelle | Licence | 4 |`
	`| E4 | Bruchez | | 5 |`
	`| E5 | Zurfluh | NULL | NULL |`
	`+-----+-------------+---------+-----------+`

Type SET

MySQL permet de rechercher des enregistrements en examinant des énumérations de type SET. Le principe est de comparer des ensembles entre eux (la liste de référence et la colonne de type SET).

Attention à l'ordre des valeurs lors de l'insertion d'une colonne de type SET, qui, s'il ne suit pas l'ordre alphabétique, ne sera pas l'ordre de stockage. L'exemple suivant illustre ce propos.

Tableau 4-25 Ordre d'insertion et de stockage d'une énumération SET

Table et données	Données stockées

```
CREATE TABLE Cursus
(num CHAR(4), nom CHAR(15),
 diplomes SET ('BTS','DUT','INSA','Licence')
DEFAULT 'DUT',
 CONSTRAINT pk_Cursus PRIMARY KEY(num));

INSERT INTO Cursus
 VALUES ('E1','Brouard',('Licence,BTS'));
INSERT INTO Cursus
 VALUES ('E2','Degrelle',('Licence,INSA,DUT'));
INSERT INTO Cursus
 VALUES ('E3','Auriol',('INSA,DUT'));
INSERT INTO Cursus
 VALUES ('E4','Zurfluh',NULL);
INSERT INTO Cursus (num,nom)
 VALUES ('E5', 'Mercier');
```

```
mysql> SELECT nom, diplomes
              FROM Cursus;

+--------------+------------------
+
| nom          |
diplomes         |
+--------------+------------------
+
| Brouard      |
BTS,Licence      |
| Degrelle     | DUT,INSA,Licence
|
| Auriol       |
DUT,INSA         |
| Zurfluh      |
NULL             |
| Mercier      |
DUT              |
+--------------+------------------
+
```

Le tableau suivant présente plusieurs extractions.

- La première requête est classique, le prédicat contient l'ensemble des valeurs qui respecte l'ordre de stockage.
- La deuxième ne retourne aucun enregistrement (`Empty set`) car l'ensemble des valeurs ne respecte pas l'ordre de stockage.
- La troisième requête, qui utilise l'opérateur `LIKE`, n'est pas recommandée, car la chaîne '`Licence`' peut être présente dans l'ensemble de référence non pas en tant qu'élément, mais en tant que sous-chaîne d'un élément.
- La quatrième requête utilise l'opérateur `FIND_IN_SET` qui convient bien au type de la colonne.
- La dernière requête utilise l'opérateur `AND` qui, appliqué à l'indice associé à chaque valeur de l'énumération, permet d'extraire les enregistrements qui incluent dans leurs diplômes la valeur `'DUT'` (associée à l'indice 2).

Tableau 4-26 Exemples d'extration d'une énumération `SET`

Requêtes	Résultats

`SELECT nom, diplomes`
`FROM Cursus`
`WHERE diplomes = 'Licence,BTS';`
	`Empty set (0.00 sec)`

`SELECT nom, diplomes`
`FROM Cursus`
`WHERE diplomes = 'BTS,Licence';`
	`+------------+-------------+`
	`| nom | diplomes |`
	`+------------+-------------+`
	`| Brouard | BTS,Licence |`
	`+------------+-------------+`

	`+-------------+------------------+` `| nom | diplomes |`

```
SELECT nom, diplomes
       FROM Cursus
       WHERE diplomes LIKE ('%Licence%');
```

```
+-------------+------------------+
| Brouard     | BTS,Licence      |
| Degrelle    | DUT,INSA,Licence |
+-------------+------------------+
```

```
SELECT nom, diplomes
      FROM Cursus
      WHERE
FIND_IN_SET('Licence',diplomes)>0;
```

```
+-------------+------------------+
| nom         | diplomes         |
+-------------+------------------+
| Brouard     | BTS,Licence      |
| Degrelle    | DUT,INSA,Licence |
+-------------+------------------+
```

```
SELECT nom, diplomes
       FROM Cursus
       WHERE diplomes & 2 ORDER BY nom;
```

```
+--------------+------------------+
| nom          | diplomes         |
+--------------+------------------+
| Degrelle     | DUT,INSA,Licence |
| Auriol       | DUT,INSA         |
| Mercier      | DUT              |
+--------------+------------------+
```

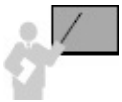
Fonctions pour les UUID

Les *Universally Unique IDentifier* (UUID, aussi appelés *Globally Unique Identifier* ou GUI) visent à identifier de façon unique un objet sans disposer nécessairement d'infrastructures de centralisation complexes. Normalisés (ISO/IEC 9834-8:2008), ces identifiants sont codés sur 128 bits et sont générés en fonction de composantes pseudo-aléatoires ainsi que des caractéristiques de l'ordinateur (numéro de disque dur, adresse MAC, etc.).

Sous sa forme textuelle, un UUID est composé de 32 caractères hexadécimaux en minuscules partitionnés en 5 groupes qui sont séparés par 4 tirets (par exemple : b353ef60-b896-11e6-b4de-e8e0b755af5d). En raisonnant en caractères, 36 sont utilisés (les 32 positions et les 4 tirets). En interne, un UUID nécessite 128 bits, soit 16 octets.

Considérons la table suivante, qui est capable de stocker pour chaque ligne deux UUID. La fonction UUID() qui retourne un UUID (conforme à la version 1 de la norme, à savoir en fonction de la date, de l'heure et de l'adrese MAC) sera utilisée dans chaque insertion.

```
CREATE TABLE bd_uuid.vols
 (id_vol SMALLINT AUTO_INCREMENT,
  uuid_varchar VARCHAR(36), uuid_bin BINARY(16),
  num_vol VARCHAR(6), date_vol DATE,
  CONSTRAINT pk_vols PRIMARY KEY(id_vol))
  DEFAULT CHARACTER SET utf8;
```



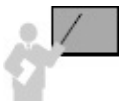
La fonction `UUID_TO_BIN (uuid_chaine [,entier))` transforme un UUID sous forme textuelle et retourne l'UUID en binaire. Cette fonction est très intéressante car elle rend davantage compacte chaque donnée dans l'éventualité d'une indexation. Utilisée avec un deuxième paramètre (valué à 1), cette fonction rend le même service en permutant les poids forts et les poids faibles.

Nous verrons plus loin l'intérêt de ce mécanisme qui a pour objectif de faire varier le moins possible les caractères du début de chaque UUID. Les insertions suivantes génèrent quatre UUID. Seule la colonne *uuid_varchar* est affichable d'un point de vue textuel.

```
INSERT INTO bd_uuid.vols (uuid_varchar, uuid_bin, num_vol, date_vol)
  VALUES (UUID(), UUID_TO_BIN(UUID()), 'AF6143', SYSDATE());
INSERT INTO bd_uuid.vols (uuid_varchar, uuid_bin, num_vol, date_vol)
 VALUES (UUID(), UUID_TO_BIN(UUID()), 'AF6147', SYSDATE());
```

Remarquez que les UID diffèrent en début d'expression pour être identiques en fin d'expression.

```
mysql> SELECT id_vol, uuid_varchar, num_vol, date_vol FROM bd_uuid.vols;
+--------+--------------------------------------+---------+------------+
| id_vol | uuid_varchar                         | num_vol | date_vol   |
+--------+--------------------------------------+---------+------------+
|      1 | 8c73b4e4-b8a4-11e6-b4de-e8e0b755af5d | AF6143  | 2016-12-02 |
|      2 | 8c74c0ff-b8a4-11e6-b4de-e8e0b755af5d | AF6147  | 2016-12-02 |
+--------+--------------------------------------+---------+------------+
```



La fonction `BIN_TO_UUID(uuid_bin [,entier))` transforme un UUID sous forme binaire et retourne l'UUID en texte. Cette fonction est très intéressante pour rendre lisible une donnée compacte. Utilisée avec un deuxième paramètre (valué à 1), elle rend le même service en permutant les poids forts et les poids faibles.

La colonne *uuid_bin* est désormais affichable d'un point de vue textuel, la différence des UUID est toujours notable en début d'expression.

```
mysql> SELECT id_vol, BIN_TO_UUID(uuid_bin), num_vol, date_vol FROM bd_
uuid.vols;
+--------+--------------------------------------+---------+------------+
| id_vol | BIN_TO_UUID(uuid_bin)                | num_vol | date_vol   |
+--------+--------------------------------------+---------+------------+
|      1 | 8c73b505-b8a4-11e6-b4de-e8e0b755af5d | AF6143  | 2016-12-02 |
|      2 | 8c74c117-b8a4-11e6-b4de-e8e0b755af5d | AF6147  | 2016-12-02 |
+--------+--------------------------------------+---------+------------+
```

L'intérêt du mécanisme de permutation des poids forts avec les poids faibles réside dans le fait de faire varier le moins possible les caractères du début de chaque UUID. Appliquer un index à de telles données est plus efficace que de l'appliquer à des données qui sont plus évolutives et augmente la fragmentation des blocs de l'index.

En conséquence et en insérant deux nouvelles lignes, les UUID seront davantage homogènes que les précédents générés sans ce principe.

```
INSERT INTO bd_uuid.vols (uuid_varchar, uuid_bin, num_vol, date_vol)
  VALUES (UUID(), UUID_TO_BIN(UUID()), 'AF6143', SYSDATE());
INSERT INTO bd_uuid.vols (uuid_varchar, uuid_bin, num_vol, date_vol)
 VALUES (UUID(), UUID_TO_BIN(UUID()), 'AF6147', SYSDATE());


mysql> SELECT id_vol, BIN_TO_UUID(uuid_bin), num_vol, date_vol
FROM bd_uuid.vols;
+--------+--------------------------------------+---------+------------+
| id_vol | BIN_TO_UUID(uuid_bin)                | num_vol | date_vol   |
+--------+--------------------------------------+---------+------------+
|      1 | 8c73b505-b8a4-11e6-b4de-e8e0b755af5d | AF6143  | 2016-12-02 |
|      2 | 8c74c117-b8a4-11e6-b4de-e8e0b755af5d | AF6147  | 2016-12-02 |
|      3 | 11e6b8a4-8c7c-fb4c-b4de-e8e0b755af5d | AF6143  | 2016-12-03 |
|      4 | 11e6b8a4-8c7e-2f64-b4de-e8e0b755af5d | AF6147  | 2016-12-03 |
+--------+--------------------------------------+---------+------------+
```

Un index peut être créé sur cette colonne de type UUID, et il sera plus optimal qu'un autre créé sur un UUID libre.

```
CREATE UNIQUE INDEX idx_vols_uuid ON bd_uuid.vols(uuid_bin);
```

Enfin, citons la fonction `IS_UUID(uuid_chaine)` qui retourne `1` si l'expression textuelle a la forme d'un UUID valide. Pour notre exemple, toutes les colonnes qui ont bénéficié du constructeur natif `UUID()` vérifient le typage.

Figure 4-5 *Vérification du type UUID*

```
mysql> SELECT id_vol, IS_UUID(uuid_varchar),
    ->                IS_UUID(BIN_TO_UUID(uuid_bin)),
    ->                     IS_UUID(num_vol)
    -> FROM   bd_uuid.vols;
+--------+----------------------+--------------------------------+-----------------+
| id_vol | IS_UUID(uuid_varchar) | IS_UUID(BIN_TO_UUID(uuid_bin)) | IS_UUID(num_vol) |
+--------+----------------------+--------------------------------+-----------------+
|      1 |                    1 |                              1 |               0 |
|      2 |                    1 |                              1 |               0 |
|      3 |                 NULL |                              1 |               0 |
|      4 |                 NULL |                              1 |               0 |
+--------+----------------------+--------------------------------+-----------------+
4 rows in set (0.00 sec)
```

Autres fonctions

D'autres fonctions n'appartenant pas à la classification précédente sont présentées dans le tableau suivant :

Tableau 4-27 Autres fonctions

Fonction	Objectif	Exemple
`DEFAULT(colonne)`	Valeur par défaut d'une colonne (`NULL` si aucune).	`SELECT DEFAULT (diplomes) FROM Cursus LIMIT 1;` retourne `'DUT'`.
`FORMAT(numerique,nb)`	Formate un nombre arrondi à *nb* décimales de la manière suivante :`'#,###,###.##'`.	`FORMAT(1234567.8901, 1)` retourne '1,234,567.9'.
`GREATEST(expression[, expression]...)`	Retourne la plus grande des expressions.	`GREATEST('Raffarin','Chirac','X-Men')` retourne `'X-Men'`.
`LEAST(expression[, expression]...)`	Retourne la plus petite des expressions.	`LEAST('Villepin','Sarkozy','X-Men')` retourne `'Sarkozy'`.

Regroupements

Cette section traite à la fois des regroupements de lignes (agrégats) et des fonctions de groupe (ou multilignes). Nous étudierons les parties surlignées de l'instruction SELECT suivante :

```
SELECT [ { DISTINCT | DISTINCTROW } | ALL ] listeColonnes
FROM nomTable [ WHERE condition ]
 [ clauseRegroupement ]
 [ HAVING condition ]
 [ clauseOrdonnancement ]
 [ LIMIT [rangDépart,] nbLignes ] ;
```

- *listeColonnes* : peut inclure des expressions (présentes dans la clause de regroupement) ou des fonctions de groupe.
- *clauseRegroupement* : GROUP BY (*expression1*[,*expression2*]...) permet de regrouper des lignes selon la valeur des expressions (colonnes, fonction, constante, calcul).
- HAVING *condition* : pour inclure ou exclure des lignes aux groupes (la condition ne peut faire intervenir que des expressions du GROUP BY).
- *ClauseOrdonnancement* : déjà étudié (ORDER BY dans la section « Projection/Ordonnancement »).

Interrogeons la table suivante en composant des regroupements et en appliquant des fonctions de groupe :

Figure 4-6 *Table Pilote*

Pilote

brevet	nom	nbHVol	prime	embauche	typeAvion	compa
PL-1	Viel	450	500	1965-02-05	A320	AF
PL-2	Donsez	0		1995-05-13	A320	AF
PL-3	Grin	1000		2001-09-11	A320	SING
PL-4	Fresnais	2450	500	2001-09-21	A330	SING
PL-5	Vielle	400	600	1965-01-16	A340	AF
PL-6	Tort		0	2000-12-24	A340	CAST

Fonctions de groupe

Nous étudions dans cette section les fonctions usuelles. D'autres sont proposées pour manipuler des cubes (*datawarehouse*).

Le tableau suivant présente les principales fonctions. L'option DISTINCT évite les duplicatas (pris en compte sinon par défaut). À l'exception de COUNT, toutes les

fonctions ignorent les valeurs NULL (il faudra utiliser IFNULL pour contrer cet effet).

Tableau 4-28 Fonctions de groupe

Fonction	Objectif
AVG([DISTINCT] *expr*)	Moyenne de *expr* (nombre).
COUNT({* | [DISTINCT] *expr*})	Nombre de lignes (* toutes les lignes, *expr* pour les colonnes non nulles).
GROUP_CONCAT(*expr*)	Composition d'un ensemble de valeurs.
MAX([DISTINCT] *expr*)	Maximum de *expr* (nombre, date, chaîne).
MIN([DISTINCT] *expr*)	Minimum de *expr* (nombre, date, chaîne).
STDDEV(*expr*)	Écart type de *expr* (nombre).
SUM([DISTINCT] *expr*)	Somme de *expr* (nombre).
VARIANCE(*expr*)	Variance de *expr* (nombre).

Utilisées sans GROUP BY, ces fonctions s'appliquent à la totalité ou à une seule partie d'une table comme le montrent les exemples suivants :

Tableau 4-29 Exemples de fonctions de groupe

Fonction	Exemples

AVG	Moyenne des heures de vol et des primes des pilotes de la compagnie 'AF'.
	`SELECT AVG(nbHVol), AVG(prime) FROM Pilote WHERE compa = 'AF';`
	`+-------------+------------+`
	`| AVG(nbHVol) | AVG(prime) |`
	`+-------------+------------+`
	`|  283.333333 |   550.0000 |`
	`+-------------+------------+`

COUNT	Nombre de pilotes, d'heures de vol et de primes (toutes et distinctes) recensées dans la table.
	`SELECT COUNT(*), COUNT(nbHVol), COUNT(prime),`
	`COUNT(DISTINCT prime) FROM Pilote;`
	`+----------+--------------+--------------+----------------------+`
	`| COUNT(*) | COUNT(nbHVol) | COUNT(prime) | COUNT(DISTINCT prime) |`
	`+----------+--------------+--------------+----------------------+`
	`|        6 |            5 |            4 |                    3 |`

	+----------+--------------+-------------+----------------------+
GROUP_CONCAT	Nom des pilotes de la compagnie 'AF'. `SELECT compa, GROUP_CONCAT(nom) FROM Pilote GROUP BY compa;` +-------+--+ | compa | GROUP_CONCAT(nom) | +-------+--+ | AF | Viel,Donsez,Vielle | | CAST | Tort | | SING | Grin,Fresnais | +-------+--
MAX – MIN	Nombre d'heures de vol le plus élevé, date d'embauche la plus récente. Nombre d'heures de vol le moins élevé, date d'embauche la plus ancienne. `SELECT MAX(nbHVol), MAX(embauche) "Date+", MIN(prime), MIN(embauche) "Date-" FROM Pilote;` +-------------+-----------+-----------+-----------+ | MAX(nbHVol) | Date+ | MIN(prime) | Date- | +-------------+-----------+-----------+-----------+ | 2450.00 | 2001-09-21 | 0 | 1965-02-05 | +-------------+-----------+-----------+-----------+
STDEV – SUM – VARIANCE	Écart type des primes, somme des heures de vol, variance des primes des pilotes de la compagnie 'AF'. `SELECT STDDEV(prime), SUM(nbHVol), VARIANCE(prime) FROM Pilote WHERE compa = 'AF';` +---------------+-------------+-----------------+ | STDDEV(prime) | SUM(nbHVol) | VARIANCE(prime) | +---------------+-------------+-----------------+ | 50.0000 | 850.00 | 2500.0000 | +---------------+-------------+----------------- +

Étudions à présent ces fonctions dans le cadre de regroupements de lignes.

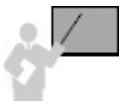
Étude du `GROUP BY` et `HAVING`

Le groupement de lignes dans une requête se programme au niveau surligné de l'instruction SQL suivante :

```
SELECT col1[, col2…], fonction1Groupe(…)[,fonction2Groupe(…)…]
 FROM nomTable [ WHERE condition ]
 GROUP BY {col1 | expr1 | position1} [,{col2... }]
 [ HAVING condition ]
[ORDER BY {col1 | expr1 | position1} [ASC | DESC] [,{col1 ... }] ];
```

- La clause `WHERE` de la requête permet d'exclure des lignes pour chaque groupement, ou de rejeter des groupements entiers. Elle s'applique donc à la totalité de la table.
- La clause `GROUP BY` liste les colonnes du groupement.

- La clause HAVING permet de poser des conditions sur chaque groupement.
- La clause ORDER BY permet de trier le résultat (déjà étudiée).



Toutes les colonnes qui doivent être présentes dans le SELECT doivent se trouver dans le GROUP BY.

Seules des fonctions d'agrégat peuvent être présentes dans le SELECT, en plus des colonnes du regroupement.

Aucun alias de colonne ne peut être utilisé dans la clause GROUP BY.

Les éventuels NULL seront regroupés dans le même ensemble de résultats (deux NULL sont considérés ici comme semblables).

Dans l'exemple suivant, en groupant sur la colonne compa, trois ensembles de lignes (groupements) sont composés. Il est alors possible d'appliquer des fonctions de groupe à chacun de ces ensembles (dont le nombre n'est pas précisé dans la requête ni limité par le système qui parcourt toute la table).

Figure 4-7 *Groupement sur la colonne* `compa`

Pilote

brevet	nom	nbHVol	prime	embauche	typeAvion	compa
PL-1	Viel	450	500	1965-02-05	A320	AF
PL-2	Donsez	0		1995-05-13	A320	AF
PL-5	Vielle	400	600	1965-01-16	A340	AF
PL-6	Tort		0	2000-12-24	A340	CAST
PL-3	Grin	1000		2001-09-11	A320	SING
PL-4	Fresnais	2450	500	2001-09-21	A330	SING

Il est aussi possible de grouper sur plusieurs colonnes (par exemple ici sur les colonnes compa et typeAvion pour classifier les pilotes selon ces deux critères).

Utilisées avec GROUP BY, les fonctions s'appliquent désormais à chaque regroupement, comme le montrent les exemples suivants :

Tableau 4-30 Exemples de fonctions de groupe avec GROUP BY

Fonction	Exemples

AVG	Moyenne des heures de vol et des primes pour chaque compagnie.
	`SELECT compa, AVG(nbHVol), AVG(prime) FROM Pilote`
	`GROUP BY(compa) ;`
	`+-------+------------+------------+`
	`| compa | AVG(nbHVol) | AVG(prime) |`
	`+-------+------------+------------+`
	`| AF | 283.333333 | 550.0000 |`
	`| CAST | NULL | 0.0000 |`
	`| SING | 1725.000000 | 500.0000 |`
	`+-------+------------+------------+`

COUNT	Nombre de pilotes (et ceux qui ont de l'expérience du vol) par compagnie.
	`SELECT compa, COUNT(*), COUNT(nbHVol) FROM Pilote`
	`GROUP BY(compa) ;`
	`+-------+----------+---------------+`
	`| compa | COUNT(*) | COUNT(nbHVol) |`
	`+-------+----------+---------------+`
	`| AF | 3 | 3 |`
	`| CAST | 1 | 0 |`
	`| SING | 2 | 2 |`
	`+-------+----------+---------------+`

MAX	Nombre d'heures de vol le plus élevé, date d'embauche la plus récente pour chaque compagnie.
	`SELECT compa, MAX(nbHVol), MAX(embauche) "Date+" FROM Pilote GROUP BY(compa) ;`
	`+-------+------------+------------+`
	`| compa | MAX(nbHVol) | Date+ |`
	`+-------+------------+------------+`
	`| AF | 450.00 | 1995-05-13 |`
	`| CAST | NULL | 2000-12-24 |`
	`| SING | 2450.00 | 2001-09-21 |`
	`+-------+------------+------------+`

STDEV – SUM (avec WHERE)	Écarts types des primes et sommes des heures de vol des pilotes volant sur 'A320' de chaque compagnie.
	`SELECT compa, STDDEV(prime), SUM(nbHVol) FROM Pilote WHERE typeAvion = 'A320' GROUP BY(compa) ;`
	`+-------+--------------+-------------+`
	`| compa | STDDEV(prime) | SUM(nbHVol) |`
	`+-------+--------------+-------------+`
	`| AF | 0.0000 | 450.00 |`
	`| SING | NULL | 1000.00 |`
	`+-------+--------------+-------------+`

	Nombre de pilotes qualifiés par type

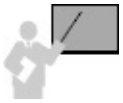
Plusieurs colonnes dans le `GROUP BY`	d'appareil et par compagnie.
	`SELECT compa,typeAvion, COUNT(brevet)`
	`FROM Pilote`
	`GROUP BY compa,typeAvion ;`
	`+-------+----------+--------------+`
	`| compa | typeAvion | COUNT(brevet) |`
	`+-------+----------+--------------+`
	`| AF | A320 | 2 |`
	`| AF | A340 | 1 |`
	`| CAST | A340 | 1 |`
	`| SING | A320 | 1 |`
	`| SING | A330 | 1 |`
	`+-------+----------+--------------+`

`GROUP BY` et `HAVING`	Compagnies (et nombre de leurs pilotes) ayant plus d'un pilote.
	`SELECT compa, COUNT(brevet) FROM Pilote`
	`GROUP BY(compa)`
	`HAVING COUNT(brevet)>=2 ;`
	`+-------+--------------+`
	`| compa | COUNT(brevet) |`
	`+-------+--------------+`
	`| AF | 3 |`
	`| SING | 2 |`
	`+-------+--------------+`

Opérateurs ensemblistes

Une des forces du modèle relationnel repose sur le fait qu'il est fondé sur une base mathématique (théorie des ensembles). Le langage SQL devrait programmer les opérations binaires (entre deux tables) suivantes :

- **intersection** qui extrait des données présentes simultanément dans les deux tables ;
- **union** par les opérateurs `UNION` et `UNION ALL` qui fusionnent des données des deux tables ;
- **différence** qui extrait des données présentes dans une table sans être présentes dans la deuxième table ;
- **produit cartésien** par le fait de combiner des jeux de résultats, soit en utilisant plusieurs tables dans la clause `FROM`, soit à l'aide de l'opérateur `CROSS JOIN` ;
- **division** par le fait de pouvoir comparer deux ensembles entre eux par une notion d'égalité.



Les comparaisons utilisées par ces opérations nécessitent de comparer des colonnes de différentes tables si elles sont de même type (CHAR, VARCHAR, DATE, numériques, etc.). Le nom des colonnes peut différer entre deux tables. Attention, pour les CHAR, vous devrez veiller à ce que la taille soit identique entre les deux tables pour que la comparaison fonctionne (avec CAST, par exemple).

L'intersection, la différence et la division ne sont pas encore implémentées par MySQL.
La différence et l'intersection peuvent se programmer à l'aide de sous-requêtes ou via des jointures. La division nécessite en plus d'utiliser l'opérateur d'existence.

Étudions à présent chaque opérateur à partir de l'exemple composé des deux tables suivantes :

- `AvionsdeAF(immat CHAR(6), typeAvion CHAR(10), nbHVol DECIMAL(10,2))`
- `AvionsdeSING(immatriculation CHAR(6), typeAv CHAR(10), prixAchat DECIMAL(14,2))`

Il est vraisemblable que seules les deux premières colonnes doivent être comparées. Bien que permises par MySQL, la programmation de l'union, l'intersection ou la différence entre les prix des avions et les heures de vol (deux colonnes numériques) ne seraient pas valides d'un point de vue sémantique.

Figure 4-8 *Tables pour les opérateurs ensemblistes*

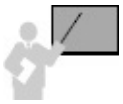
AviondeAF

immat	typeAvion	nbHVol
F-WTSS	Concorde	6570
F-GLFS	A320	3500
F-GTMP	A340	

AviondeSING

immatriculation	typeAv	PrixAchat
S-ANSI	A320	104 500
S-AVEZ	A320	156 000
S-SMILE	A330	198 000
F-GTMP	A340	204 500

Intersection



L'intersection entre deux ensembles homogènes se programme à l'aide d'une requête du type SELECT DISTINCT *ensemble1* FROM *Table1* WHERE *ensemble1* IN (SELECT *ensemble2* FROM *Table2*). Comme l'opérateur d'intersection, cette requête est commutative (*Table1* peut jouer le rôle de *Table2*, et *ensemble1* celui d'*ensemble2*).

Notez qu'à l'affichage le nom des colonnes est donné par la première requête. La deuxième fait apparaître deux colonnes dans le SELECT.

Tableau 4-31 Exemples d'intersections

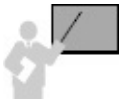
Besoin	Requête

Quels sont les types d'avions que les deux compagnies exploitent en commun?	`SELECT DISTINCT typeAvion FROM AvionsdeAF`
	`  WHERE  typeAvion IN`
	`  (SELECT typeAv FROM AvionsdeSING);`
	`+-----------+`
	`| typeAvion |`
	`+-----------+`
	`| A320  |`
	`| A340  |`
	`+-----------+`

Quels sont les avions qui sont exploités par les deux compagnies en commun?	`SELECT DISTINCT immatriculation ,typeAv`
	`  FROM AvionsdeSING`
	`  WHERE immatriculation IN`
	`  (SELECT immat FROM AvionsdeAF)`
	`  AND typeAv IN`
	`  (SELECT typeAvion FROM AvionsdeAF);`
	`+-----------------+--------+`
	`| immatriculation | typeAv |`
	`+-----------------+--------+`
	`| F-GTMP  | A340  |`
	`+-----------------+--------+`

Si vous voulez continuer ce raisonnement en vous basant sur trois compagnies, il suffit d'ajouter une clause WHERE dans les requêtes imbriquées qui interrogera la troisième compagnie. Ce principe se généralise, et, pour *n* compagnies, il faudra *n* imbrications de requêtes.

Opérateurs UNION et UNION ALL



Les opérateurs UNION et UNION ALL sont commutatifs. L'opérateur UNION permet d'éviter les duplicatas (comme DISTINCT dans un SELECT). UNION ALL ne les élimine pas.

Tableau 4-32 Exemples avec les opérateurs UNION

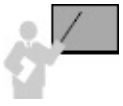
Besoin	Requête

Quels sont tous les types d'avions que les deux compagnies exploitent ?	`SELECT typeAvion FROM AvionsdeAF`
	`UNION`
	`SELECT typeAv FROM AvionsdeSING;`
	`+-----------+`
	`| typeAvion |`
	`+-----------+`
	`| A320 |`
	`| A340 |`
	`| Concorde |`
	`| A330 |`
	`+-----------+`

Même requête avec les duplicatas. On extrait les types de la compagnie 'AF', suivis des types de la compagnie 'SING'.	`SELECT typeAvion FROM AvionsdeAF`
	`UNION ALL`
	`SELECT typeAv FROM AvionsdeSING;`
	`+-----------+`
	`| typeAvion |`
	`+-----------+`
	`| A320 |`
	`| A340 |`
	`| Concorde |`
	`| A340 |`
	`| A320 |`
	`| A320 |`
	`| A330 |`
	`+-----------+`

Ce principe se généralise à l'union de *n* ensembles par *n* requêtes reliées avec *n-1* clauses UNION ou UNION ALL.

Différence



La différence entre deux ensembles homogènes se programme à l'aide d'une requête du type SELECT DISTINCT *ensemble1* FROM *Table1* WHERE *ensemble1* NOT IN (SELECT *ensemble2* FROM *Table2*). Comme l'opérateur ensembliste, cette requête n'est pas commutative, car elle programme *ensemble1-ensemble2*.

Tableau 4-33 Exemples de différences

Besoin	Requête

Quels sont les types d'avions exploités par la compagnie 'AF', mais pas par 'SING' ?	`SELECT DISTINCT typeAvion FROM AviondeAF`
	`  WHERE typeAvion NOT IN`
	`  (SELECT typeAv FROM AviondeSING);`
	`+----------+`
	`| typeAvion |`
	`+----------+`
	`| Concorde  |`
	`+----------+`

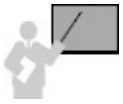
Quels sont les types d'avions exploités par la compagnie 'SING,' mais pas par 'AF' ?	`SELECT DISTINCT typeAv FROM AviondeSING`
	`  WHERE typeAv NOT IN`
	`  (SELECT typeAvion  FROM AviondeAF);`
	`+--------+`
	`| typeAv |`
	`+--------+`
	`| A330   |`
	`+--------+`

Ce principe se généralise à la différence entre *n* ensembles par imbrication de *n* requêtes (dans le bon ordre).

La directive `NOT IN` doit être utilisée avec prudence car elle retourne *faux* si un membre ramené par la sous-interrogation est `NULL`.

Produit cartésien

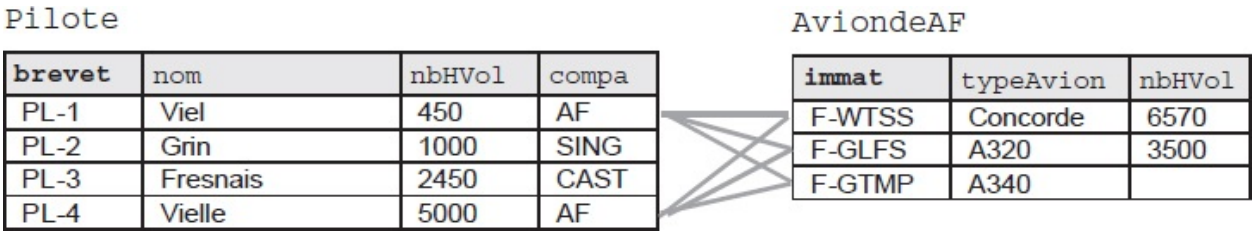
En mathématiques, le produit cartésien de deux ensembles *E* et *F* est l'ensemble des couples (*x*, *y*) où $x \in E$ et $y \in F$. En transposant au modèle relationnel, le produit cartésien de deux tables *T1* et *T2* est l'ensemble des enregistrements (*x*, *y*) où $x \in T1$ et $y \in T2$.



Le produit cartésien entre deux tables *T1* et *T2* se programme soit à l'aide de l'opérateur `CROSS JOIN`, soit en positionnant les deux tables dans le `FROM` sans condition de correspondance dans le `WHERE`.

Le produit cartésien restreint, illustré par l'exemple suivant, exprime les combinaisons d'équipages qu'il est possible de réaliser en considérant les pilotes de la compagnie 'AF' et les avions de la table `AviondeAF`.

Figure 4-9 *Produit cartésien d'enregistrements de tables*



Le nombre d'enregistrements résultant d'un produit cartésien est égal au produit du nombre d'enregistrements des deux tables mises en relation.

Dans le cadre de notre exemple, le nombre d'enregistrements du produit cartésien sera de 2 pilotes x 3 avions = 6 enregistrements. Le tableau suivant décrit les requêtes équivalentes pour produire un produit cartésien restreint. Les alias distinguent les colonnes s'il advenait qu'il en existe de même nom entre les deux tables.

Tableau 4-34 Produit cartésien

Besoin et requêtes	Résultat

Quels sont les couples possibles (*avion*, *pilote*) en considérant les avions et les pilotes de la compagnie 'AF' ?
`SELECT p.brevet, avAF.immat`
`FROM Pilote p CROSS JOIN AvionsdeAF avAF`
`WHERE p.compa = 'AF';`
`SELECT p.brevet, avAF.immat`
`FROM Pilote p, AvionsdeAF avAF`
`WHERE p.compa = 'AF';`
	`+--------+--------+`
	`| brevet | immat |`
	`+--------+--------+`
	`| PL-1 | F-GLFS |`
	`| PL-1 | F-GTMP |`
	`| PL-1 | F-WTSS |`
	`| PL-4 | F-GLFS |`
	`| PL-4 | F-GTMP |`
	`| PL-4 | F-WTSS |`
	`+--------+--------+`

Division

La division permet de comparer un ensemble avec un autre ensemble (dit de référence). La division permet par exemple de répondre aux questions suivantes : « quels sont les pilotes qui détiennent toutes les qualifications existantes ? », « dans quel magasin trouve-t-on les mêmes produits dans le rayon jardinage que dans le magasin Bricomarché de Castanet-Tolosan ? ». Vient alors une nuance concernant la comparaison. Si l'égalité doit être parfaite, alors on parle de division exacte et les magasins qui disposent davantage de produits que celui de Castanet ne seront pas extraits. S'il s'agit

d'inclusion, alors on parle de division inexacte et les magasins mieux dotés que le magasin de référence respectent la condition.

Une division peut se programmer de nombreuses manières à l'aide de regroupements et de sous-requêtes, mais la solution la plus élégante pour comparer deux ensembles est de tester la non-existence de deux différences.

Définition

Considérons l'exemple suivant pour décrire la requête à construire. Il s'agit de répondre à la question : « *Quels sont les avions affrétés par* toutes *les compagnies françaises ?* ». L'ensemble de référence (*A*) est constitué des codes des compagnies françaises. L'ensemble à comparer (*B*) est formé des codes des compagnies pour chaque avion.

Deux cas sont à envisager suivant la manière de comparer les deux ensembles :

- Division inexacte (le reste n'est pas nul) : un ensemble est seulement inclus dans un autre ($A \in B$). La question à programmer serait : « *Quels sont les avions affrétés par* toutes *les compagnies françaises ?* », sans préciser si les avions ne doivent pas être aussi affrétés par des compagnies étrangères. L'avion ('A3', 'Mercure') répondrait à cette question, que la dernière ligne de la table `Affretements` soit présente ou pas.
- Division exacte (le reste est nul) : les deux ensembles sont égaux ($B=A$). La question à programmer serait : « *Quels sont les avions affrétés* exactement *(ou* uniquement*) par toutes les compagnies françaises ?* ». L'avion ('A3', 'Mercure') répondrait à cette question si la dernière ligne de la table `Affretements` était inexistante. Les lignes concernées dans les deux tables sont grisées.

***Figure 4-10** Divisions à programmer*

Affretements

immat	typeAv	compa	dateAff
A1	A320	SING	1965-05-13
A2	A340	AF	1968-06-22
A3	Mercure	AF	1965-02-05
A4	A330	ALIB	1965-01-16
A3	Mercure	ALIB	1942-03-05
A3	Mercure	SING	1987-03-01

Compagnie

comp	nomComp	pays
AF	Air France	F
ALIB	Air Lib	F
SING	Singapore AL	SG

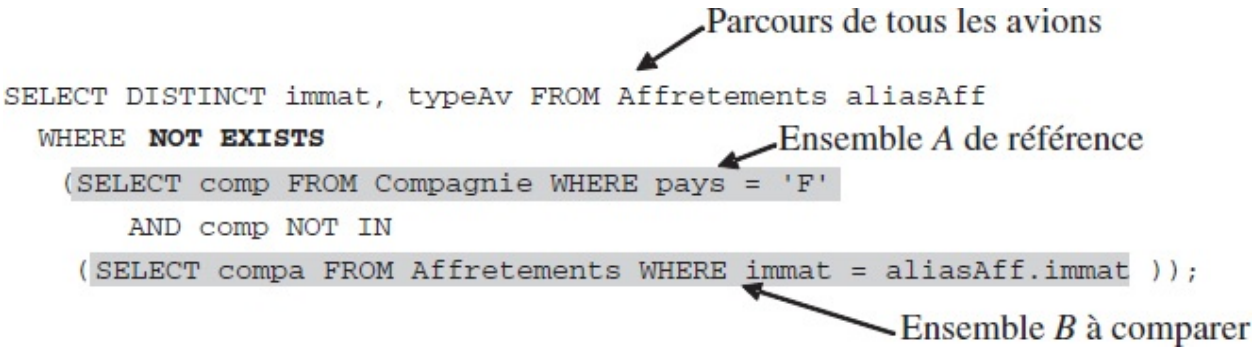
Résultat

immat	typeAv
A3	Mercure

L'opérateur de différence (programmé avec NOT IN) combiné à la fonction EXISTS permet de programmer ces deux comparaisons (un ensemble inclus dans un autre et une égalité d'ensembles). Il existe d'autres solutions à base de regroupements et de sous-interrogations (synchronisées ou pas) que nous n'étudierons pas, parce qu'elles me semblent plus compliquées. Écrivons à présent ces deux divisions à l'aide de requêtes SQL.

Division inexacte

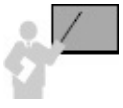
Pour programmer le fait qu'un ensemble est seulement inclus dans un autre (ici $A \subset B$), il faut qu'il n'existe pas d'élément dans l'ensemble $\{A\text{-}B\}$. La différence se programme à l'aide de l'opérateur NOT IN, l'inexistence d'élément se programme à l'aide de la fonction NOT EXISTS comme le montre la requête suivante :



Division exacte

Pour programmer le fait qu'un ensemble est strictement égal à un autre (ici $A=B$), il faut qu'il n'existe ni d'élément dans l'ensemble $\{A\text{-}B\}$ ni dans l'ensemble $\{B\text{-}A\}$. La traduction mathématique est la suivante : $A=B \Leftrightarrow (A\text{-}B=\varnothing$ et $B\text{-}A=\varnothing)$. Les opérateurs se programment de la même manière que pour la requête précédente. Le « et » se programme avec un AND.

Ordonner des résultats



Le résultat d'une requête contenant des opérateurs ensemblistes est trié, par défaut, par ordre croissant, sauf avec l'opérateur UNION ALL.

La clause ORDER BY n'est utilisable qu'une fois en fin d'une requête incluant des opérateurs ensemblistes. Cette clause accepte le nom des colonnes de la première requête ou la position de ces colonnes.

Le tableau suivant présente trois écritures différentes de la même requête ensembliste contenant une clause ORDER BY. Le besoin est de connaître tous les types d'avions que les deux compagnies exploitent (classement par ordre décroissant).

Notez que la troisième requête produit le même résultat en faisant intervenir un SELECT (aliasé) dans le FROM. Ce mécanisme permet de construire dynamiquement la table à interroger.

Tableau 4-35 Exemples avec la clause ORDER BY

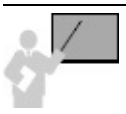
Technique	Requête

Nom de la colonne.	`SELECT typeAvion FROM AvionsdeAF`
	`UNION`
	`SELECT typeAv FROM AvionsdeSING`
	`ORDER BY typeAvion DESC;`

Position de la colonne.	`...` `ORDER BY 1 DESC;`

SELECT dans le FROM.	`SELECT T.typeAvion FROM`
	`  (SELECT typeAvion FROM AvionsdeAF UNION`
	`  SELECT typeAv FROM AvionsdeSING) T`
	`ORDER BY T.typeAvion DESC;`

	`+-----------+`
	`| typeAvion |`
	`+-----------+`
	`| Concorde  |`
	`| A340      |`
	`| A330      |`
	`| A320      |`
	`+-----------+`



Il faut affecter des alias aux expressions de la première requête pour pouvoir les utiliser dans le ORDER BY final.

Pour illustrer cette restriction, supposons que nous désirions faire la liste des avions avec leur prix d'achat augmenté de 20 %, liste triée en fonction de cette dernière hausse. Le problème est que la table AviosdeAF ne possède pas une telle colonne. Ajoutons donc au SELECT de cette table, dans le tableau suivant, la valeur 0 pour rendre possible l'opérateur UNION.

Tableau 4-36 Alias pour ORDER BY

Requête	Résultat

```
SELECT immatriculation,
  1.2*prixAchat px
FROM AvionsdeSING
UNION
  SELECT immat, 0 FROM AvionsdeAF
ORDER BY px DESC;
```

```
+-----------------+------------+
| immatriculation | px         |
+-----------------+------------+
| F-GTMP          | 245400.000 |
| S-MILE          | 237600.000 |
| S-AVEZ          | 187200.000 |
| S-ANSI          | 125400.000 |
| F-GLFS          |      0.000 |
| F-GTMP          |      0.000 |
| F-WTSS          |      0.000 |
+-----------------+------------+
```

Parcours de tous les avions

```
SELECT DISTINCT immat, typeAv FROM Affretements aliasAff
  WHERE NOT EXISTS
    (SELECT comp FROM Compagnie WHERE pays = 'F  '
                AND comp NOT IN
        (SELECT compa FROM Affretements WHERE immat = aliasAff.immat ))
  AND NOT EXISTS
    (SELECT compa FROM Affretements WHERE immat = aliasAff.immat
                AND compa NOT IN
        (SELECT comp FROM Compagnie WHERE pays = 'F' ));
```

A–B

B–A

Bilan

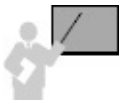
Dès lors que vous travaillerez en termes ensemblistes, vous devrez toujours manipuler des colonnes de même type. Pour rendre homogène des jeux de résultats, vous pourrez utiliser dans les sous-requêtes des expressions (constantes ou calculs).

Tout NULL extrait de vos sous-requêtes est comparable à un autre NULL (ce qui n'est en général pas vrai dans une comparaison traditionnelle, à savoir non

ensembliste). Ce principe est également suivi pour les regroupements (`GROUP BY`) où les `NULL` composent un unique ensemble.

Jointures

Les jointures permettent d'extraire des données issues de plusieurs tables. Le processus de normalisation du modèle relationnel est basé sur la décomposition et a pour conséquence d'augmenter le nombre de tables d'un schéma. Ainsi, la majorité des requêtes utilisent des jointures nécessaires pour pouvoir extraire des données de tables distinctes.



Une jointure met en relation deux tables sur la base d'une clause de jointure (comparaison de colonnes). Généralement, cette comparaison fait intervenir une clé étrangère d'une table avec une clé primaire d'une autre table (car le modèle relationnel est fondamentalement basé sur les valeurs).

En considérant les tables suivantes, les seules jointures logiques doivent se faire sur l'égalité soit des colonnes `comp` et `compa` soit des colonnes `brevet` et `chefPil`. Ces jointures permettront d'afficher des données d'une table (ou des deux tables) tout en posant des conditions sur une table (ou les deux). Par exemple, l'affichage du nom des compagnies (colonne de la table `Compagnie`) qui ont embauché un pilote ayant moins de 500 heures de vol (condition sur la table `Pilote`).

Figure 4-11 *Deux tables à mettre en jointure*

Compagnie

comp	nrue	rue	ville	nomComp
AF	124	Port Royal	Paris	Air France
SING	7	Camparols	Singapour	Singapore AL
CAST	1	G. Brassens	Blagnac	Castanet AL

Pilote

brevet	nom	nbHVol	compa	chefPil
PL-1	Lamothe	450	AF	PL-4
PL-2	Linxe	900	AF	PL-4
PL-3	Soutou	1000	SING	
PL-4	Alquié	3400	AF	

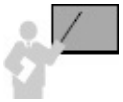
Classification

Une jointure peut s'écrire, dans une requête SQL, de différentes manières :

- « relationnelle » (aussi appelée « SQL89 » pour rappeler la version de la norme SQL) ;
- « SQL2 » (aussi appelée « SQL92 ») ;
- « procédurale » (qui qualifie la structure de la requête) ;
- « mixte » (combinaison des trois approches précédentes).

Nous allons principalement étudier les deux premières écritures qui sont les plus utilisées. Nous parlerons en fin de section des deux dernières.

Jointure relationnelle



La forme la plus courante de la jointure est la jointure dite « relationnelle » (aussi appelée SQL89), caractérisée par une seule clause FROM contenant les tables et alias à mettre en jointure deux à deux. La syntaxe générale suivante décrit une jointure relationnelle :

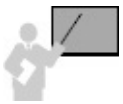
```
SELECT [alias1.]col1, [alias2.]col2…
    FROM [nomBase.]nomTable1 [alias1], [nomBase.]nomTable2 [alias2]…
```

```
    WHERE (conditionsDeJointure);
```

Cette forme est la plus utilisée, car elle est la plus simple à écrire. Un autre avantage de ce type de jointure est qu'elle laisse le soin au SGBD d'établir la meilleure stratégie d'accès (choix du premier index à utiliser, puis du deuxième, etc.) pour optimiser les performances.

Afin d'éviter les ambiguïtés concernant le nom des colonnes, on utilise en général des alias de tables pour suffixer les tables dans la clause FROM et préfixer les colonnes dans les clauses SELECT et WHERE.

Jointures SQL2



Afin de se rendre conforme à la norme SQL2, MySQL propose aussi des directives qui permettent de programmer d'une manière plus verbale les différents types de jointures :

```
SELECT [ALL | DISTINCT | DISTINCTROW ] listeColonnes
 FROM [nomBase.]nomTable1 [{ INNER | { LEFT | RIGHT } [OUTER] }]
  JOIN [nomBase.]nomTable2{ ON condition | USING ( colonne1 [, colonne2]... )}
  | { CROSS JOIN | NATURAL [{ LEFT | RIGHT } [OUTER] ]
      JOIN [nomBase.]nomTable2 } …
[ WHERE condition ];
```

Cette écriture est moins utilisée que la syntaxe relationnelle. Bien que plus concise pour des jointures à deux tables, elle se complique pour des jointures plus complexes.

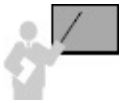
Types de jointures

Bien que, dans le vocabulaire courant, on ne parle que de « jointures » en fonction de la nature de l'opérateur utilisé dans la requête, de la clause de jointure et des tables concernées, on distingue :

- Les jointures internes (*inner joins*) :
  - L'équijointure (*equi join*) est la plus connue, elle utilise l'opérateur

d'égalité dans la clause de jointure. La jointure naturelle est conditionnée en plus par le nom des colonnes. La non équijointure utilise l'opérateur d'inégalité dans la clause de jointure.

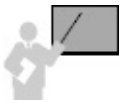
  - L'autojointure (*self join*) est un cas particulier de l'équijointure, qui met en œuvre deux fois la même table (des alias de tables permettront de distinguer les enregistrements entre eux).
  - L'inéquijointure met en relation des lignes de plusieurs tables sans utiliser aucune égalité entre clés primaires (ou candidates) et clés étrangères.
- La jointure externe (*outer join*) réalise à la fois une jointure interne et permet de relier des lignes sans correspondance. Cette opération favorise une table (dite « dominante ») par rapport à l'autre (dite « subordonnée »).



Pour mettre trois tables *T1*, *T2* et *T3* en jointure, il faut utiliser deux clauses de jointures (une entre *T1* et *T2* et l'autre entre *T2* et *T3*). Pour *n* tables, il faut *n*-1 clauses de jointures. Si vous oubliez une clause de jointure, un produit cartésien restreint est généré.

Étudions à présent chaque type de jointure avec les syntaxes « relationnelle » et « SQL2 ».

Équijointure



Une équijointure utilise l'opérateur d'égalité dans la clause de jointure et compare généralement des clés primaires avec des clés étrangères.

En considérant les tables suivantes, les équijointures se programment soit sur les colonnes `comp` et `compa` soit sur les colonnes `brevet` et `chefPil`. Extrayons par exemple :

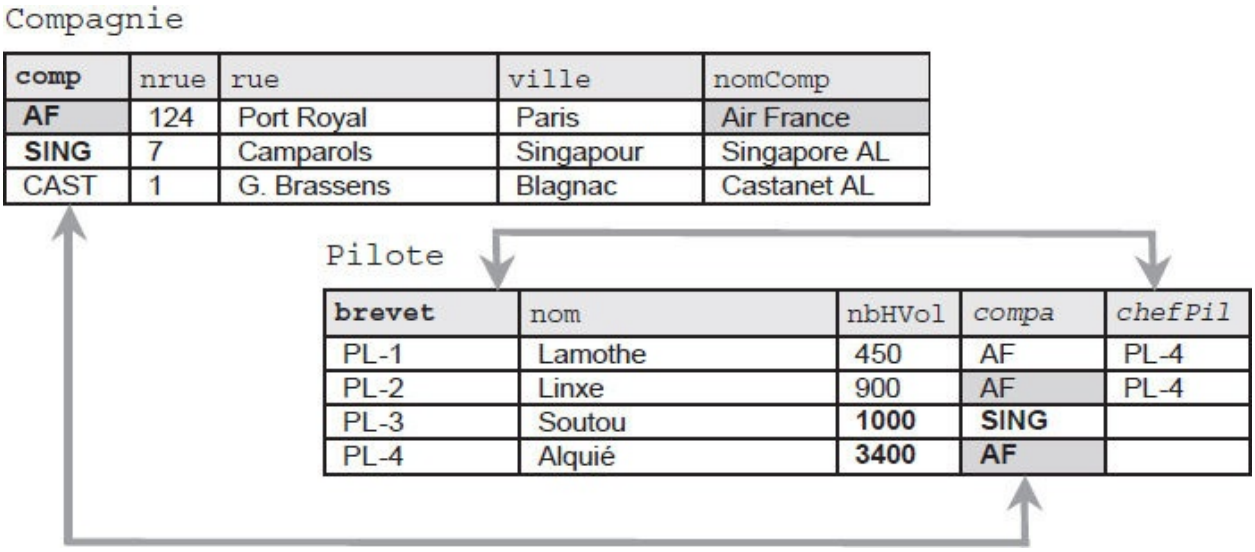
- l'identité des pilotes de la compagnie de nom 'Air France' ayant plus de

500 heures de vol (requête *R1*) ;

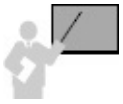
- les coordonnées des compagnies qui embauchent des pilotes de moins de 500 heures de vol (requête *R2*).

La jointure qui résoudra la première requête est illustrée dans la figure par les données grisées, tandis que la deuxième jointure est représentée par les données en gras.

Figure 4-12 *Équijointures*

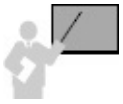


Écriture « relationnelle »



- MySQL recommande d'utiliser des alias de tables pour améliorer les performances.
- Les alias sont obligatoires quand des colonnes de différentes tables portent le même nom ou dans le cas d'autojointures.

Écriture « SQL2 »



- La clause `JOIN … ON condition` permet de programmer une équijointure.

- L’utilisation de la directive `INNER` devant `JOIN…` est optionnelle et est appliquée par défaut.

Tableau 4-37 Exemples d’équijointures

Requête	Jointure relationnelle	Jointure SQL2

R1	`SELECT brevet, nom`
	` FROM  Pilote, Compagnie`
	` WHERE comp    = compa`
	` AND   nomComp = 'Air France'`
	` AND   nbHVol  > 500;`
		`SELECT brevet, nom`
		` FROM  Compagnie`
		` JOIN Pilote ON comp = compa`
		` WHERE nomComp = 'Air France'`
		` AND   nbHVol  > 500;`

	`+--------+---------+`
	`| brevet | nom     |`
	`+--------+---------+`
	`| PL-2   | Linxe   |`
	`| PL-4   | Alquié  |`
	`+--------+---------+`	

R2	`SELECT cpg.nomComp, cpg.nrue,`
	`  cpg.rue, cpg.ville`
	` FROM  Pilote pil, Compagnie cpg`
	` WHERE cpg.comp    = pil.compa`
	` AND   pil.nbHVol  < 500;`
		`SELECT  nomComp, nrue, rue, ville`
		` FROM Compagnie`
		` INNER JOIN Pilote ON comp = compa`
		` WHERE nbHVol  < 500;`

	`+------------+------+------------+-------+`
	`| nomComp    | nrue | rue        | ville |`
	`+------------+------+------------+-------+`
	`| Air France |  124 | Port Royal | Paris |`
	`+------------+------+------------+-------+`	

Autojointure

Cas particulier de l’équijointure, l’autojointure relie une table à elle même.

Extrayons par exemple :

- l’identité des pilotes placés sous la responsabilité des pilotes de nom 'Alquié' (requête *R3*) ;
- la somme des heures de vol des pilotes placés sous la responsabilité des chefs pilotes de la compagnie de nom 'Air France' (requête *R4*).

Ces requêtes doivent être programmées à l’aide d’une autojointure, car elles imposent de parcourir deux fois la table `Pilote` (examen de chaque pilote en le comparant à un autre). Les autojointures sont réalisées entre les colonnes `brevet` et `chefPil`.

La jointure de la première requête est illustrée dans la figure par les données surlignées en clair, tandis que la deuxième jointure est mise en valeur par les données surlignées en foncé.

***Figure 4-13** Autojointures*

Le tableau suivant détaille ces requêtes, les clauses d'autojointures sont surlignées. Dans les deux syntaxes, il est impératif d'utiliser un alias de table. Concernant l'écriture « SQL2 », on remarque que les clauses `JOIN` s'imbriquent (pour joindre plus de deux tables).

Tableau 4-38 Exemples d'autojointures

Requête	Jointure relationnelle	Jointure SQL2

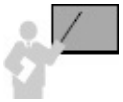
R3	`SELECT p1.brevet, p1.nom`
	` FROM Pilote p1, Pilote p2`
	` WHERE p1.chefPil = p2.brevet`
	` AND p2.nom = 'Alquié';`
		`SELECT p1.brevet, p1.nom`
		` FROM Pilote p1`
		`  JOIN Pilote p2`
		`  ON p1.chefPil = p2.brevet`
		` WHERE p2.nom = 'Alquié';`

	`+--------+---------+`
	`| brevet | nom |`
	`+--------+---------+`
	`| PL-1 | Lamothe |`
	`| PL-2 | Linxe |`
	`+--------+---------+`	

R4	`SELECT SUM(p1.nbHVol)`
	` FROM Pilote p1, Pilote p2,`
	`  Compagnie cpg`
	` WHERE p1.chefPil = p2.brevet`
	` AND cpg.comp = p2.compa`
	` AND cpg.nomComp = 'Air France';`
		`SELECT SUM(p1.nbHVol)`
		` FROM Pilote p1`
		` JOIN Pilote p2`
		` ON p1.chefPil = p2.brevet`
		`  JOIN Compagnie`
		`  ON comp = p2.compa`
		`  WHERE nomComp = 'Air France';`

	`+----------------+`
	`| SUM(p1.nbHVol) |`
	`+----------------+`
	`| 1350.00 |`
	`+----------------+`	

Inéquijointure



Les requêtes d'inéquijointures font intervenir tout type d'opérateur (`<>`, `>`, `<`, `>=`, `<=`, `BETWEEN`, `LIKE`, `IN`). À l'inverse des équijointures, la clause d'une inéquijointure n'est pas basée sur l'égalité de clés primaires (ou candidates) et de clés étrangères.

En considérant les tables suivantes, extrayons par exemple :

- les pilotes ayant plus d'expérience que le pilote de numéro de brevet 'PL-2' (requête *R5*).
- le titre de qualification des pilotes en raisonnant sur la comparaison des heures de vol avec un ensemble de référence, ici la table `HeuresVol` (requête *R6*). Dans notre exemple, il s'agit par exemple de retrouver le fait que le premier pilote est débutant.

La jointure qui résoudra la deuxième requête est illustrée par les niveaux de gris.

Figure 4-14 *Inéquijointures*

Pilote

brevet	nom	nbHVol	*compa*	*chefPil*
PL-1	Lamothe	**450**	AF	PL-4
PL-2	Linxe	900	AF	PL-4
PL-3	Soutou	1000	SING	*NULL*
PL-4	Alquié	3400	AF	*NULL*

HeuresVol

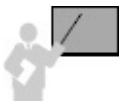
titre	basnbHVol	hautnbHVol
Débutant	**0**	**500**
Initié	501	1000
Expert	1001	20000

Le tableau suivant détaille ces requêtes, les clauses d'inéquijointures sont surlignées :

Tableau 4-39 Exemples d'inéquijointures

Requête	Jointure relationnelle	Jointure SQL2
R5	`SELECT p1.brevet, p1.nom, p1.nbHVol, p2.nbHVol "Référence" FROM Pilote p1, Pilote p2 WHERE p1.nbHVol > p2.nbHVol AND p2.brevet = 'PL-2';`	`SELECT p1.brevet, p1.nom, p1.nbHVol, p2.nbHVol "Référence" FROM Pilote p1 JOIN Pilote p2 ON p1.nbHVol>p2.nbHVol WHERE p2.brevet = 'PL-2';`
	`+--------+------------------+---------+-----------+ | brevet | nom | nbHVol | Référence | +--------+------------------+---------+-----------+ | PL-3 | Soutou | 1000.00 | 900.00 | | PL-4 | Alquié | 3400.00 | 900.00 | +--------+------------------+---------+-----------+`	
R6	`SELECT pil.brevet, pil.nom, pil.nbHVol, hv.titre FROM Pilote pil, HeuresVol hv WHERE pil.nbHVol BETWEEN hv.basnbHVol AND hv.hautnbHVol;`	`SELECT brevet, nom, nbHVol, titre FROM Pilote JOIN HeuresVol ON nbHVol BETWEEN basnbHVol AND hautnbHVol;`
	`+--------+------------------+---------+----------+ | brevet | nom | nbHVol | titre | +--------+------------------+---------+----------+ | PL-1 | Lamothe | 450.00 | Débutant | | PL-2 | Linxe | 900.00 | Initié | | PL-3 | Soutou | 1000.00 | Initié | | PL-4 | Alquié | 3400.00 | Expert | +--------+------------------+---------+----------+`	

Jointures externes



Une jointure externe est une jointure interne qui inclut en plus les lignes qui ne sont pas en correspondance (et qui ne remontent pas avec la seule jointure interne du fait de NULL dans une clé étrangère ou de l'absence de correspondance de valeur dans l'autre table).

Lorsque deux tables sont à joindre de manière externe, vous pourrez choisir une table pour laquelle vous désirez obtenir les lignes sans correspondance. Cette table est dite « dominante » par rapport à l'autre qui est dite « subordonnée ». Dans le jeu de résultat, en regard des lignes de la table dominante, les colonnes des lignes de la table subordonnée sont toutes NULL (même la clé primaire). On distingue les jointures unilatérales qui considèrent une table dominante et une table subordonnée, et les jointures bilatérales pour lesquelles les deux tables jouent un rôle symétrique (pas de dominant).

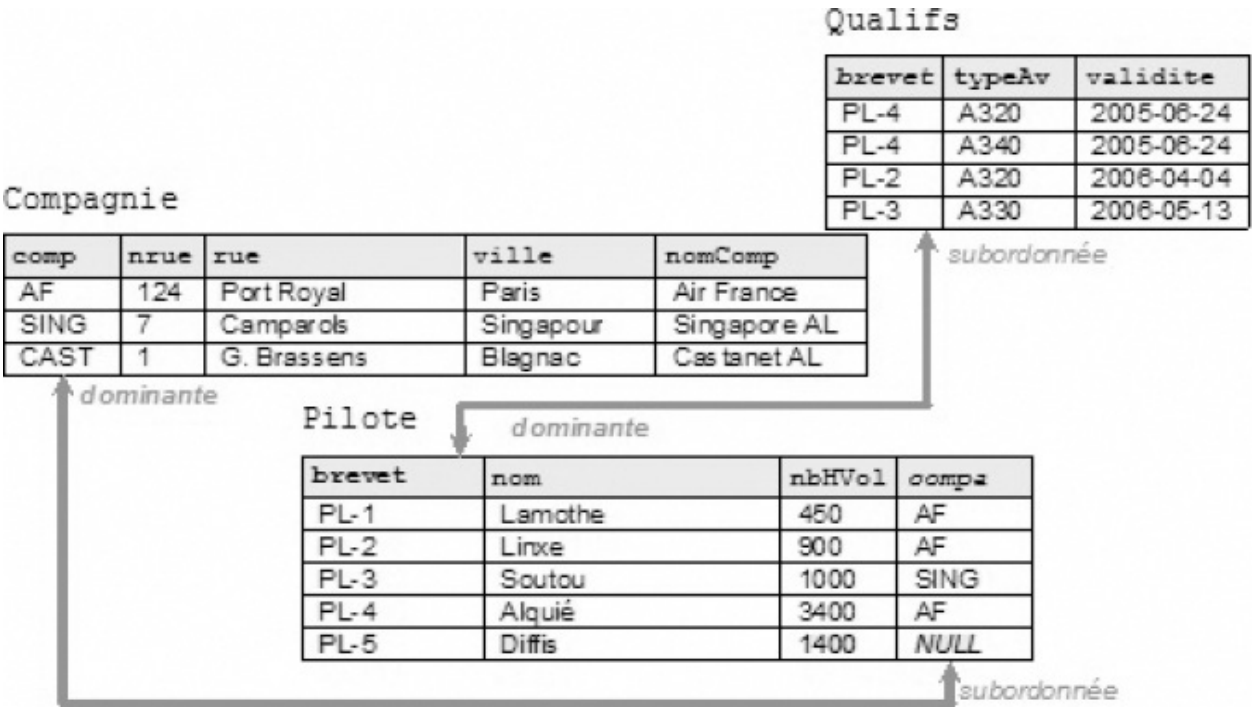
Jointures unilatérales

En considérant les tables suivantes, une jointure externe unilatérale permet d'extraire :

- les compagnies qui n'embauchent aucun pilote (requête *R7*) ;
- les pilotes ne disposant d'aucune qualification (requête *R8*).

La figure illustre les tables dominantes et subordonnées :

Figure 4-15 *Jointures externes unilatérales*



Écriture « SQL2 »

Vous désignerez la table dominante dans la clause `OUTER JOIN` avec l'option `LEFT` ou `RIGHT`. C'est cette table pour laquelle vous allez extraire davantage de lignes. Pour chaque ligne extraite sans correspondance de la table dominante, les colonnes de l'autre table seront systématiquement à `NULL`. Je vous conseille d'afficher les clés des deux tables dans vos requêtes pour mieux comprendre le sens de la jointure.

Le tableau suivant présente les deux écritures équivalentes. Pour isoler les lignes sans correspondance, il suffit de tester la nullité de la clé de la table subordonnée.

Tableau 4-40 Écritures équivalentes de jointures externes unilatérales

Requête	Jointures relationnelles	Jointures SQL2

R7	Sans objet.	`SELECT c.comp, c.nomComp, p.brevet, p.nom`
		`FROM Compagnie c`
		`**LEFT OUTER JOIN** Pilote p ON c.comp = p.compa`
		`WHERE p.brevet IS NULL;`
		`-- équivalent à`
		`SELECT c.comp, c.nomComp, p.brevet, p.nom`
		`FROM Pilote p`
		`**RIGHT OUTER JOIN** Compagnie c ON c.comp = p.compa`
		`WHERE p.brevet IS NULL;`

		`+------+------------+--------+------+`
		`| comp | nomComp | brevet | nom |`
		`+------+------------+--------+------+`
		`| CAST | Castanet AL | **NULL** | **NULL** |`
		`+------+------------+--------+------+`

R8	Sans objet.	`SELECT q.brevet, q.typeAv, p.brevet, p.nom`
		`FROM Qualifs q`
		`**RIGHT OUTER JOIN** Pilote p ON p.brevet = q.brevet`
		`WHERE q.brevet IS NULL;`
		`-- idem`
		`SELECT q.brevet, q.typeAv, p.brevet, p.nom`
		`FROM Pilote p`
		`**LEFT OUTER JOIN** Qualifs q ON p.brevet = q.brevet`
		`WHERE q.brevet IS NULL;`

		`+--------+--------+--------+---------+`
		`| brevet | typeAv | brevet | nom |`
		`+--------+--------+--------+---------+`
		`| **NULL** | **NULL** | PL-1 | Lamothe |`
		`| **NULL** | **NULL** | PL-5 | Diffis |`
		`+--------+--------+--------+---------+`

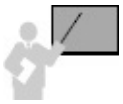
Jointures bilatérales

Dans ce type de jointure, les deux tables jouent un rôle symétrique : il n’y a pas de table dominante. Cela permet d’extraire des lignes qui ne sont en correspondance ni d’un côté, ni de l’autre. En considérant les tables précédentes, une jointure externe bilatérale permettrait d’extraire à la fois les compagnies qui n’embauchent aucun pilote et les pilotes qui ne sont rattachés à aucune compagnie.

MySQL n'a pas encore implémenté la jointure externe bilatérale avec FULL OUTER JOIN. En conséquence, il faudra mettre en œuvre ce mécanisme à l'aide de l'union de deux jointures externes unilatérales.

Pour mettre en œuvre la jointure bilatérale sans FULL OUTER JOIN, vous devrez réaliser l'union des deux jointures externes bilatérales d'une manière symétrique. Dans l'exemple, il s'agirait d'extraire à la fois des pilotes sans compagnie et les compagnies sans pilote. Le tableau suivant présente ce type d'émulation.

Jointures procédurales



Les jointures procédurales sont écrites par des requêtes qui contiennent des sous-interrogations (SELECT imbriqué). Chaque clause FROM ne contient qu'une seule table.

```
SELECT colonnesTable1 FROM [nomBase.]nomTable1
    WHERE colonne(s) | expression(s) { IN | = | opérateur }
          (SELECT colonne(s)delaTable2 FROM [nomBase.]nomTable2
             WHERE colonne(s) | expression(s) { IN | = | opérateur }
                  (SELECT …)
             [AND (conditionsTable2)]
           )
    [AND (conditionsTable1)];
```

Cette forme de jointure est plus complexe à écrire car il est nécessaire de respecter l'ordre des tables dans les clauses FROM pour exprimer chaque parcours. Il est souvent nécessaire d'avoir recours à ce type de jointure lorsqu'une des sous-requêtes contient un regroupement ou un calcul particulier.

Seules les colonnes de la table qui se trouve au niveau du premier SELECT peuvent être extraites.

La sous-interrogation doit être placée entre parenthèses. Elle ne doit pas comporter de clause ORDER BY, mais peut inclure GROUP BY et HAVING.

Le résultat d'une sous-interrogation est utilisé par la requête de niveau supérieur. Une sous-interrogation est exécutée avant la requête de niveau supérieur.

Une sous-interrogation peut ramener une ou plusieurs lignes. Les opérateurs =, >, <, >=, <= permettent d'en extraire une, les opérateurs IN, ANY et ALL permettent d'en ramener plusieurs.

Sous-interrogations monolignes

Le tableau suivant détaille quelques sous-interrogations monolignes. Nous nous basons sur certaines requêtes déjà étudiées (forme relationnelle et SQL2).

Tableau 4-41 Sous-interrogations monolignes

Opérateur	Besoin	Requête

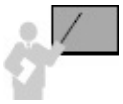
= pour les équijointures ou autojointures (= teste une ligne)	*R1* (Pilotes de la compagnie de nom 'Air France' ayant plus de 500 heures de vol.)	`SELECT brevet, nom FROM Pilote`
		` WHERE compa =`
		`  (SELECT comp`
		`  FROM Compagnie`
		`  WHERE nomComp = 'Air France')`
		` AND nbHVol>500;`

	R3 (Pilotes sous la responsabilité du pilote de nom 'Alquié'.)	`SELECT brevet, nom FROM Pilote`
		` WHERE chefPil =`
		`  (SELECT brevet FROM Pilote`
		`  WHERE nom LIKE '%Alquié%');`

> pour les inéquijointures	*R5* (Pilotes ayant plus d'expérience que le pilote de brevet 'PL-2'.)	`SELECT brevet, nom, nbHVol FROM Pilote`
		` WHERE nbHVol >`
		`  (SELECT nbHVol FROM Pilote`
		`  WHERE brevet = 'PL-2');`

Sous-interrogations multilignes (IN, ALL et ANY)

Les opérateurs multilignes sont les suivants :



- IN compare un élément à une donnée quelconque d'une liste ramenée par la sous-interrogation. Cet opérateur est utilisé pour les équijointures et les autojointures (et les intersections). L'opérateur NOT IN sera employé pour les jointures externes (et les différences).
- ANY compare l'élément à chaque donnée ramenée par la sous-interrogation. L'opérateur « =ANY » équivaut à IN. L'opérateur « <ANY » signifie « inférieur à au moins une des valeurs », donc « inférieur au maximum ». L'opérateur « >ANY » signifie « supérieur à au moins une des valeurs », donc « supérieur au minimum ».
- ALL compare l'élément à tous ceux ramenés par la sous-interrogation. L'opérateur « <ALL » signifie « inférieur au minimum » et « >ALL » signifie « supérieur au maximum ».

Le tableau suivant détaille quelques sous-interrogations multilignes. Le dernier exemple programme une partie d'une jointure externe.

Tableau 4-42 Sous-interrogations multilignes

Opérateur	Besoin	Requête

IN	*R2.* Coordonnées des compagnies qui embauchent des pilotes de moins de 500 heures de vol.	`SELECT nomComp, nrue, rue, ville`
		` FROM Compagnie`
		` WHERE comp IN`
		`  (SELECT compa FROM Pilote`
		`  WHERE nbHVol < 500);`

= et IN	*R4.* Somme des heures de vol des pilotes placés sous la responsabilité des chefs pilotes de la compagnie de nom 'Air France'.	`SELECT SUM(nbHVol) FROM Pilote`
		` WHERE chefPil IN`
		`  (SELECT brevet FROM Pilote`
		`  WHERE compa =`
		`  (SELECT comp FROM Compagnie`
		`  WHERE nomComp = 'Air France')`
		`  );`

NOT IN	Compagnies n'ayant pas de pilote.	`SELECT  nomComp, nrue, rue, ville`
		` FROM Compagnie`
		` WHERE comp NOT IN`
		`  (SELECT compa FROM Pilote`
		`  WHERE compa IS NOT NULL);`

L'opérateur NOT IN doit être utilisé avec prudence car il retourne *FALSE* dès le premier NULL retourné par la sous-requête. Pour contrecarrer cet effet de bord, vous devrez utiliser dans la sous-requête une des fonctions suivantes : IFNULL, IS NULL, IS NOT NULL ou COALESCE.

Afin d'illustrer les opérateurs ANY et ALL, considérons la table suivante. Nous avons indiqué en gras les nombres d'heures minimal et maximal des A320, en grisé les nombres d'heures minimal et maximal des avions de la compagnie 'AF'.

***Figure 4-16** Table Avion*

Avions

immat	typeAv	nbHVol	compa
A1	A320	**1000**	AF
A2	A330	1500	AF
A3	A320	**550**	SING
A4	A340	1800	SING
A5	A340	200	AF
A6	A330	100	AF

Le tableau suivant détaille quelques jointures procédurales utilisant les opérateurs ALL et ANY :

Tableau 4-43 Opérateurs ALL et ANY

Opérateur	Besoin	Requête et résultat

ANY	*R11*. Avions dont le nombre d'heures de vol est inférieur à celui de n'importe quel A320.	`SELECT immat, typeAv, nbHVol`
		`FROM Avion`
		`WHERE nbHVol < ANY`
		`(SELECT nbHVol FROM Avion`
		`WHERE typeAv='A320');`
		`+-------+--------+--------+`
		`| immat | typeAv | nbHVol |`
		`+-------+--------+--------+`
		`| A3 | A320 | 550.00 |`
		`| A5 | A340 | 200.00 |`
		`| A6 | A330 | 100.00 |`
		`+-------+--------+--------+`

	R12. Compagnies et leurs avions dont le nombre d'heures de vol est supérieur à	`SELECT immat, typeAv, nbHVol, compa`
		`FROM Avion`
		`WHERE nbHVol > ANY`
		`(SELECT nbHVol FROM Avion`
		`WHERE compa = 'SING');`
		`+-------+--------+---------+-------+`

	celui de n'importe quel avion de la compagnie de code 'SING'.	`| immat | typeAv | nbHVol  | compa |`
		`+-------+--------+---------+-------+`
		`| A1    | A320   | 1000.00 | AF    |`
		`| A2    | A330   | 1500.00 | AF    |`
		`| A4    | A340   | 1800.00 | SING  |`
		`+-------+--------+---------+-------+`

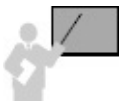
ALL	*R13*. Avions dont le nombre d'heures de vol est inférieur à tous les A320.	`SELECT  immat, typeAv, nbHVol`
		`  FROM Avion`
		`  WHERE nbHVol < ALL`
		`  (SELECT nbHVol FROM Avion`
		`  WHERE typeAv='A320');`
		`+-------+--------+--------+`
		`| immat | typeAv | nbHVol |`
		`+-------+--------+--------+`
		`| A5    | A340   | 200.00 |`
		`| A6    | A330   | 100.00 |`
		`+-------+--------+--------+`

	R14. Compagnies et leurs avions dont le nombre d'heures de vol est supérieur à tous les avions de la compagnie de code 'AF'.	`SELECT  immat, typeAv, nbHVol, compa`
		`  FROM Avion`
		`  WHERE nbHVol > ALL`
		`  (SELECT nbHVol FROM Avion`
		`  WHERE compa = 'AF');`
		`+-------+--------+---------+-------+`
		`| immat | typeAv | nbHVol  | compa |`
		`+-------+--------+---------+-------+`
		`| A4    | A340   | 1800.00 | SING  |`
		`+-------+--------+---------+-------+`

Tables dérivées (et CTE)

Une table dérivée est une table qui est utilisée comme une source de données, construite à l'occasion d'une requête donnée. Initialement ce mécanisme était naturellement mis en œuvre dans la clause `FROM` de la requête.

Depuis, la technique des CTE (*Common Table Expressions*) est apparue. Elle consiste à déclarer les tables dérivées en amont de la requête principale. Comme une vue à usage unique, une CTE est une source de données qui sera interrogée soit dans la requête principale à l'aide d'un alias de table, soit dans une autre CTE qui sera définie après. L'utilisation de CTE facilite l'écriture de requêtes complexes. L'atout majeur des CTE est la possibilité de mettre en œuvre la récursivité (voir en fin de chapitre).



Une requête peut utiliser plusieurs tables dérivées en tant que sources de données. Ces tables dérivées seront exploitables à l'aide d'un alias de table.

```
SELECT listeColonnes
FROM   (SELECT… FROM table1…) alias1
     [,(SELECT… FROM table2…) alias2 …]
[ WHERE …]
[ ORDER BY …];
```

Considérons la table suivante. Le but est d'extraire le pourcentage partiel de pilotes par compagnie. Dans notre exemple, il y a 5 pilotes dont 3 dépendent de 'AF'. Pour cette compagnie le pourcentage partiel de pilotes est de 3/5 soit 60 %.

***Figure 4-17** Table Pilote*

Pilote

brevet	nom	nbHVol	compa
PL-1	Lamothe	450	AF
PL-2	Linxe	900	AF
PL-3	Soutou	1000	SING
PL-4	Alquié	3400	AF
PL-5	Castaings		

La requête suivante construit deux tables dérivées (alias `a` et `b`) dans la clause `FROM` pour répondre à cette question :

Tableau 4-44 Tables dérivées

Requête et tables évaluées dans le `FROM`	Résultat

```
SELECT IFNULL(a.compa,' ') AS "Comp",
       FORMAT(a.nbpil/b.total*100,1) AS "%Pilote"
  FROM (SELECT compa, COUNT(*) AS nbpil
              FROM Pilote GROUP BY compa)     a,
       (SELECT COUNT(*) AS total FROM Pilote)  b;
```

a

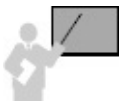
compa	nbpil
AF	3
SING	1
NULL	1

b

total
5

```
+------+---------+
| Comp | %Pilote |
+------+---------+
|      | 20.0    |
| AF   | 60.0    |
| SING | 20.0    |
+------+---------+
```

La mise en œuvre des tables dérivées avec la technique des CTE est plus légère car elle simplifie la clause `FROM` tout en permettant aux CTE d'être éventuellement visibles entre elles.



Prévues dans une version 8 de MySQL, les CTE seront définies dans la clause WITH qui affectera autant d'alias qu'il y a de tables dérivées. Ces alias de tables seront ensuite exploités dans la requête finale.

```
WITH
    cte1 AS (SELECT… FROM table1…)
  [,cte2 AS (SELECT… FROM table2…)]
  [,…]
SELECT listeColonnes
[ WHERE …]
[ ORDER BY …];
```

La requête précédente s'écrira à l'aide d'une CTE non récursive (ici Maria DB 10.2 est utilisée, MySQL 8.0 n'implémentant pas encore ce mécanisme).

Figure 4-18 *CTE non récursive*

```
MariaDB> WITH
       compte_comp AS (SELECT compa, COUNT(*) AS nbpil FROM  Pilote GROUP BY compa),
       total_comp  AS (SELECT COUNT(*) AS total FROM Pilote)
       SELECT IFNULL(compte_comp.compa,' ') AS Comp,
              FORMAT(compte_comp.nbpil/total_comp.total*100,1) AS "%Pilote"
       FROM   compte_comp, total_comp
       ORDER BY Comp;
+------+---------+
| Comp | %Pilote |
+------+---------+
|      | 20.0    |
| AF   | 60.0    |
| SING | 20.0    |
+------+---------+
```

Sous-interrogations synchronisées

Une sous-interrogation est synchronisée si elle manipule des colonnes d'une table du niveau supérieur. Une sous-interrogation synchronisée est exécutée une fois pour chaque enregistrement extrait par la requête de niveau supérieur. Cette technique peut être aussi utilisée dans les ordres UPDATE et DELETE. La forme générale d'une sous-interrogation synchronisée est la suivante. Les alias des tables sont utiles pour pouvoir manipuler des colonnes de tables de différents niveaux.

```
SELECT alias1.c
  FROM nomTable1 alias1
  WHERE colonne(s) opérateur (SELECT alias2.z…
                              FROM  nomTable2 alias2
                              WHERE  alias1.x opérateur alias2.y)
  [AND (conditionsTable1 )];
```

Une sous-interrogation synchronisée peut ramener une ou plusieurs lignes. Différents opérateurs peuvent être employés (=, >, <, >=, <=, EXISTS).

Opérateur mathématique

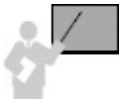
Le tableau suivant détaille un exemple d'opérateur mathématique appliqué à une sous-interrogation synchronisée :

Tableau 4-45 Sous-interrogation synchronisée

Besoin	Requête et résultat

R15. Avions dont le nombre d'heures de vol est supérieur au nombre d'heures de vol moyen des avions de leur compagnie (ici 700 heures pour 'AF' et 1115 heures pour 'SING').	`SELECT avi1.* FROM Avion avi1`
	`WHERE avi1.nbHVol >`
	`(SELECT AVG(avi2.nbHVol) FROM Avion avi2`
	`WHERE avi2.compa = avi1.compa);`
	
	`+-------+--------+---------+-------+`
	`| immat | typeAv | nbHVol | compa |`
	`+-------+--------+---------+-------+`
	`| A1 | A320 | 1000.00 | AF |`
	`| A2 | A330 | 1500.00 | AF |`
	`| A4 | A340 | 1800.00 | SING |`
	`+-------+--------+---------+-------+`

Opérateur EXISTS



L'opérateur EXISTS permet d'interrompre la sous-interrogation dès le premier enregistrement trouvé. La valeur FALSE est retournée si aucun enregistrement n'est extrait par la sous-interrogation.

Utilisons la table suivante pour décrire l'utilisation de l'opérateur EXISTS :

Figure 4-19 *Utilisation de EXISTS*

Pilote

brevet	nom	nbHVol	compa	*chefPil*
PL-1	Lamothe	450	AF	
PL-2	Linxe	900	AF	PL-4
PL-3	Soutou	1000	SING	PL-4
PL-4	Alquié	3400	AF	
PL-5	Castaings			

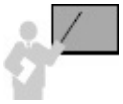
La sous-interrogation synchronisée est surlignée dans le script suivant :

Tableau 4-46 Opérateur EXISTS

Besoin	Requête et résultat

R15. Pilotes ayant au moins un pilote sous leur responsabilité.	`SELECT pil1.brevet, pil1.nom, pil1.compa FROM Pilote pil1 WHERE EXISTS (SELECT pil2.* FROM Pilote pil2 WHERE pil2.chefPil = pil1.brevet);`
	`+--------+-------------+-------+`
	`| brevet | nom | compa |`
	`+--------+-------------+-------+`
	`| PL-4 | Alquié | AF |`
	`+--------+-------------+-------+`

Opérateur NOT EXISTS



L'opérateur NOT EXISTS retourne la valeur TRUE si aucun enregistrement n'est extrait par la sous-interrogation. Cet opérateur peut être utilisé pour écrire des jointures externes.

Autres directives SQL2

Étudions enfin les autres options des jointures SQL2 (NATURAL JOIN, et USING).

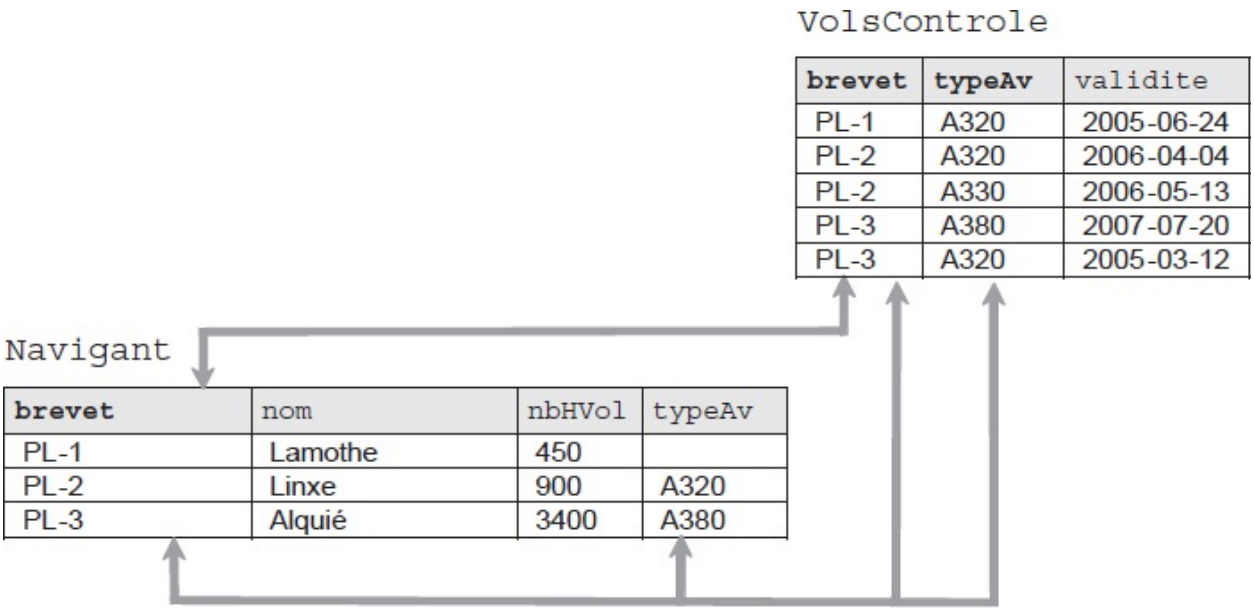
Tableau 4-47 Opérateur NOT EXISTS

Besoin	Requête et résultat

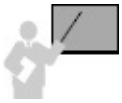
Liste des compagnies n'ayant pas de pilote.	`SELECT cpg.* FROM Compagnie cpg WHERE NOT EXISTS (SELECT compa FROM PiloteWHERE compa = cpg.comp);`
	`+------+------+------------+---------+------------+`
	`| comp | nrue | rue | ville | nomComp |`
	`+------+------+------------+---------+------------+`
	`| CAST | 1 | G. Brassens | Blagnac | Castanet AL |`
	`+------+------+------------+---------+------------+`

Considérons le schéma suivant (des colonnes portent le même nom). La colonne typeAv dans la table Navigant désigne le type d'appareil sur lequel le pilote est instructeur.

Figure 4-20 *Deux tables à mettre en jointure naturelle*



Opérateur `NATURAL JOIN`



La jointure naturelle est programmée par la clause `NATURAL JOIN`. La clause de jointure est automatiquement construite sur la base de toutes les colonnes portant le même nom entre les deux tables.

Ce type de jointure n'est pas conseillé car il ne permet pas de choisir les colonnes de jointures et, si des colonnes de même nom viennent à être ajoutées dans plusieurs tables, la jointure portera automatiquement sur ces nouvelles colonnes.

Le tableau suivant détaille deux écritures possibles d'une jointure naturelle. La clause de jointure est basée sur les colonnes (`brevet`, `typeAv`). Une clause `WHERE` aurait pu aussi être progammée.

Tableau 4-48 Jointures naturelles

Besoin	Jointures SQL2 et résultat

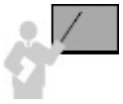
Navigants qualifiés sur un type d'appareil et instructeurs sur ce	`SELECT brevet, nom, typeAv, validite`
	` FROM Navigant NATURAL JOIN VolsControle;`
	`/* équivalent à */`

même type.

```
SELECT brevet, nom, typeAv, validite
 FROM VolsControle NATURAL JOIN Navigant;
```

```
+--------+-------------+--------+------------+
| brevet | nom         | typeAv | validite   |
+--------+-------------+--------+------------+
| PL-2   |  Linxe      | A320   | 2006-04-04 |
| PL-3   |  Alquié     | A380   | 2007-07-20 |
+--------+-------------+--------+------------+
```

Opérateur `USING`



La directive `USING(col1, col2…)` de la clause `JOIN` programme une jointure naturelle restreinte à un ensemble de colonnes. Il ne faut pas utiliser d'alias de tables dans la liste des colonnes.

Ce type de jointure est limitatif puisqu'il impose que les colonnes clés des différentes tables portent le même nom.

Dans notre exemple, on peut restreindre la jointure naturelle aux colonnes `brevet` ou `typeAv`. Si on les positionnait (`brevet`, `typeAv`) dans la directive `USING`, cela reviendrait à construire un `NATURAL JOIN`. Le tableau suivant détaille deux écritures d'une jointure naturelle restreinte :

Tableau 4-49 Jointures naturelles restreintes

Besoin	Jointures SQL2 et résultat

Nom des navigants avec leurs qualifications et dates de validité.	`SELECT nom, v.typeAv, v.validite FROM Navigant JOIN VolsControle v USING(brevet);`
	`/* équivalent à */`
	`SELECT nom, v.typeAv, v.validite FROM VolsControle v JOIN Navigant USING(brevet);`

```
+----------------+--------+------------+
| nom            | typeAv | validite   |
+----------------+--------+------------+
| Lamothe        | A320   | 2005-06-24 |
| Linxe          | A320   | 2006-04-04 |
| Linxe          | A330   | 2006-05-13 |
| Alquié         | A380   | 2007-07-20 |
| Alquié         | A320   | 2005-03-12 |
+----------------+--------+------------+
```

Les collations et jeux de caractères

Étudions à présent comment la directive `COLLATE` permet de composer avec des jeux de caractères et des collations différentes dans une comparaison ou un tri.

Tableau 4-50 Table associée à plusieurs collations

Définition de la table	Contenu

`CREATE TABLE bdutil.Compagnie`
`(comp VARCHAR(6),`
`ville VARCHAR(30) CHARACTER SET`
`latin1 COLLATE`
`latin1_general_cs,`
`nom_comp VARCHAR(20) CHARACTER SET`
`latin1 COLLATE`
`latin1_general_ci`
`) DEFAULT CHARACTER SET`
`utf8 COLLATE utf8_bin;`
	`mysql> SELECT *`
	`FROM bdutil.Compagnie;`
	`+------+----------+------------+`
	`| comp | ville | nom_comp |`
	`+------+----------+------------+`
	`| AF | Paris | Air France |`
	`| af | paris | air france |`
	`| ae | agen | air Agen |`
	`| àé | AGEN | AIR AGEN |`
	`| ab | béziers | air 34 |`
	`| AB | beziers | AIR 34 |`
	`+------+----------+------------+`

Pour ne pas distinguer la casse et l'accentuation de la colonne `comp` qui est de collation binaire (héritée de la table), il faudra adopter une collation insensible à ces deux aspects (ici `utf8_general_ci`).

Tableau 4-51 Ne pas distinguer la casse et l'accentuation

Utilisation de la collation de la colonne	Utilisation d'une collation particulière

`mysql> SELECT DISTINCT comp`
`FROM bdutil.Compagnie;`
`+------+`
`| comp |`
`+------+`
`| AF |`
`| af |`
`| ae |`
`| àé |`
`| ab |`
`| AB |`
`+------+`
	`mysql> SELECT DISTINCT comp`
	`COLLATE utf8_general_ci`
	`FROM bdutil.Compagnie;`
	`+------------------------------+`
	`| comp COLLATE utf8_general_ci |`
	`+------------------------------+`
	`| AF |`
	`| ae |`
	`| ab |`
	`+------------------------------+`

Concernant les regroupements, `COLLATE` permet de contrôler le classement des caractères pour choisir de grouper ou pas des chaînes de caractères. Dans l'exemple suivant, la collation permet d'ignorer la casse mais pas les accents (toutes les collations latines sont sensibles aux accents).

Tableau 4-52 Regroupements avec COLLATE

Utilisation de la collation de la colonne	Utilisation d'une collation particulière

`mysql> SELECT ville,`
`COUNT(*) AS nb_compagnies`
`FROM bdutil.Compagnie`
`GROUP BY ville;`
`+----------+---------------+`
`| ville | nb_compagnies |`
`+----------+---------------+`
`| AGEN | 1 |`
`| agen | 1 |`
`| beziers | 1 |`
`| béziers | 1 |`
`| Paris | 1 |`
`| paris | 1 |`
`+----------+---------------+`
	`mysql> SELECT UPPER(ville) AS ville,`
	`COUNT(*) AS nb_compagnies`
	`FROM bdutil.Compagnie`
	`GROUP BY ville`
	`COLLATE latin1_general_ci;`
	`+----------+---------------+`
	`| ville | nb_compagnies |`
	`+----------+---------------+`
	`| AGEN | 2 |`
	`| BEZIERS | 1 |`
	`| BÉZIERS | 1 |`
	`| PARIS | 2 |`
	`+----------+---------------+`

Pour les tris, deux écritures sont possibles en fonction de l'existence d'un alias de colonne. Les deux tris suivants sont basés sur une collation binaire (ils ordonnent d'abord les lettres majuscules avant les minuscules et sont sensibles aux accents).

Tableau 4-53 Regroupements avec `ORDER BY`

Utilisation de la collation de la colonne	Utilisation d'une collation particulière

```
mysql> SELECT comp, ville, nom_comp
       FROM   bdutil.Compagnie
       ORDER BY ville
                COLLATE latin1_bin;
+------+----------+------------+
| comp | ville    | nom_comp   |
+------+----------+------------+
| àé   | AGEN     | AIR AGEN   |
| AF   | Paris    | Air France |
| ae   | agen     | air Agen   |
| AB   | beziers  | AIR 34     |
| ab   | béziers  | air 34     |
| af   | paris    | air france |
+------+----------+------------+
```

```
mysql> SELECT comp
              COLLATE utf8_bin AS code,
              ville, nom_comp
       FROM   bdutil.Compagnie
       ORDER BY code;
+------+----------+------------+
| code | ville    | nom_comp   |
+------+----------+------------+
| AB   | beziers  | AIR 34     |
| AF   | Paris    | Air France |
| ab   | béziers  | air 34     |
| ae   | agen     | air Agen   |
| af   | paris    | air france |
| àé   | AGEN     | AIR AGEN   |
+------+----------+------------+
```

En forçant la collation dans une condition, il est possible de gérer la casse et l'accentuation dans des filtres.

Tableau 4-54 Collation avec `WHERE`

Utilisation d'une collation particulière	Résultat

```
SELECT comp, ville, nom_comp
FROM   bdutil.Compagnie
WHERE  ville
       COLLATE latin1_general_ci = 'PARIS'
AND    nom_comp = 'AIR FRANCE'
ORDER BY comp;
```

```
+------+-------+------------+
| comp | ville | nom_comp   |
+------+-------+------------+
| AF   | Paris | Air France |
| af   | paris | air france |
+------+-------+------------+
```

Enfin, la directive `COLLATE` peut également s'appliquer dans des conditions de regroupements (`HAVING`), dans des comparaisons de formats (`LIKE`) et dans des fonctions d'agrégats (`MAX`, `MIN`…).

Récursivité avec les CTE

Afin de parcourir un arbre ou un graphe, il est nécessaire d'utiliser la clause `WITH` accompagnée de la directive `RECURSIVE` et de l'union de deux CTE. La première CTE (appelée *anchor member* ou *seed select*) désigne le point de départ (un nœud précis pour un arbre ou un graphe). La deuxième CTE (appelée *recursive member*) permet de relier les nœuds connectés au premier,et ainsi de suite. Une jointure est nécessaire pour cela. Vous ne devrez pas utiliser `DISTINCT`, `GROUP BY` ni de sous-requête dans la deuxième CTE.

```
WITH RECURSIVE arbre_graphe [(…)] AS
```

```
     (requete_inititalisation_du_parcours
      UNION ALL
      requete_jointure_avec_arbre_graphe)
SELECT … FROM arbre_graphe [WHERE  … ]
```

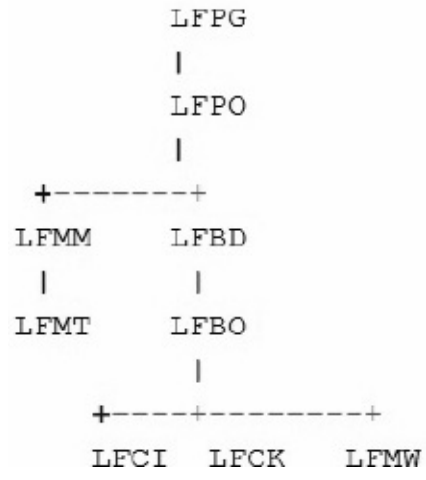
L'implémentation de la récursivité via les CTE n'est pas encore assurée par MySQL 8.0, mais cette technique est opérationnelle depuis Maria DB 10.2.2 (qui est un *fork* de MySQL et qui permet de préfigurer les nouvelles fonctionnalités de MySQL).

La table qui va nous servir d'exemple décrit une hiérarchie entre aéroports. L'association reflexive est mise en œuvre par la clé étrangère `OACI_resp` qui désigne, pour chaque aéroport, l'aéroport du niveau supérieur.

Table

```
MariaDB> SELECT OACI, nom_aero, OACI_resp
         FROM Aeroport;
+------+------------------------+-----------+
| OACI | nom_aero               | OACI_resp |
+------+------------------------+-----------+
| LFPG | Paris Charles de Gaule | NULL      |
| LFPO | Paris Orly             | LFPG      |
| LFBO | Toulouse Blagnac       | LFBD      |
| LFBD | Bordeaux Merignac      | LFPO      |
| LFCI | ALbi                   | LFBO      |
| LFCK | Castres                | LFBO      |
| LFMW | Castelnaudary          | LFBO      |
| LFMT | Montpellier Fregorgues | LFMM      |
| LFMM | Marseille Marignane    | LFPO      |
+------+------------------------+-----------+
```

Arbre modélisé



Parcours d'un arbre

La requête suivante parcourt l'arbre des aéroports en largeur d'abord puis en profondeur ensuite (sens de parcours par défaut). Le point de départ est l'aéroport d'Orly à partir duquel chaque descendant est recherché successivement selon le même procédé, à savoir une jointure avec la ligne parente. L'affichage de l'indentation s'opère avec la fonction `LPAD` qui insère des espaces à chaque nouveau niveau.

Tableau 4-55 CTE récursive pour parcourir un arbre

Requête	Résultat

```
WITH RECURSIVE sous_Paris_Orly
     (OACI, OACI_resp, niveau)
AS (SELECT OACI, OACI_resp, 0 AS niveau
       FROM Aeroport WHERE OACI = 'LFPO'
    UNION ALL
    SELECT a.OACI, a.OACI_resp, niveau+1
       FROM sous_Paris_Orly sp INNER JOIN
          Aeroport a ON sp.OACI =
a.OACI_resp)
SELECT   OACI_resp, LPAD(OACI,4*niveau,'
'),
niveau
FROM     sous_Paris_Orly
WHERE    niveau > 0
ORDER BY niveau, OACI;
```

```
+-----------+-------------+--------+
| OACI_resp | descendants  | niveau |
+-----------+-------------+--------+
| LFPO      | LFBD         |      1 |
| LFPO      | LFMM         |      1 |
| LFBD      |     LFBO     |      2 |
| LFMM      |     LFMT     |      2 |
| LFBO      |         LFCI |      3 |
| LFBO      |         LFCK |      3 |
| LFBO      |         LFMW |      3 |
+-----------+-------------+--------+
```

Seul le parcours d'un arbre en largeur d'abord puis en profondeur risque d'être proposé dans les versions 8 de MLySQL qui implémenteront le mécanisme de récursivité. Il n'est donc pas encore possible de parcourir un arbre en profondeur d'abord puis en largeur.

Constituer une liste des ascendants

La requête suivante parcourt l'arbre en partant de l'aéroport d'Orly et, pour chaque descendant trouvé, la liste des descendants est complétée.

Tableau 4-56 CTE récursive pour extraire les ascendants

Requête	Résultat

```
WITH RECURSIVE sous_Paris_Orly
 (OACI, OACI_resp, niveau, aero_liste) AS
  (SELECT OACI, OACI_resp, 0 AS niveau,
          CAST(OACI_resp AS CHAR(30))
          FROM Aeroport WHERE OACI =
'LFPO'
  UNION ALL
  SELECT a.OACI, a.OACI_resp, niveau+1,
    CAST(CONCAT(aero_liste,',',a.OACI_resp)
               AS CHAR(30))
    FROM sous_Paris_Orly sp INNER JOIN
        Aeroport a ON sp.OACI =
a.OACI_resp)
SELECT   OACI, niveau, aero_liste
FROM     sous_Paris_Orly
```

```
+------+--------+--------------------+
| OACI | niveau | aero_liste         |
+------+--------+--------------------+
| LFBD |      1 | LFPG,LFPO          |
| LFMM |      1 | LFPG,LFPO          |
| LFBO |      2 | LFPG,LFPO,LFBD     |
| LFMT |      2 | LFPG,LFPO,LFMM     |
| LFCI |      3 | LFPG,LFPO,LFBD,LFBO |
| LFCK |      3 | LFPG,LFPO,LFBD,LFBO |
| LFMW |      3 | LFPG,LFPO,LFBD,LFBO |
+------+--------+--------------------+
```

```
WHERE    niveau > 0
ORDER BY niveau, OACI;
```

Parcours d'un graphe orienté

Parcourir un graphe est similaire à parcourir un arbre : cela nécessite de la récursivité. Pour les graphes orientés, vous devrez utiliser `DISTINCT` pour ne choisir qu'un point de départ dans la requête d'initialisation.

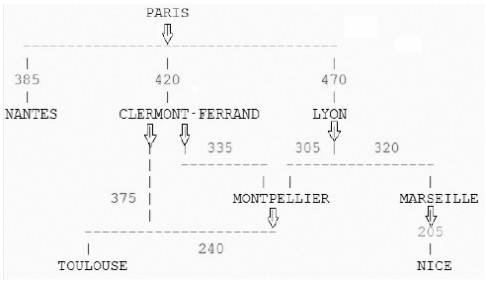
La table qui va nous servir d'exemple décrit des connexions entre villes (exemple issu de Frédéric Brouard dans http://sqlpro.developpez.com/cours/sqlserver/cte-recursives/). L'association reflexive est mise en œuvre par le couple de clés (*ville_de*, *ville_vers*) qui désigne chaque segment du graphe qui est orienté du fait de la sémantique différente des clés.

Table

```
MariaDB> SELECT ville_de, ville_vers, km
         FROM Autoroutes;
+------------------+------------------+-------+
| ville_de         | ville_vers       | km    |
+------------------+------------------+-------+
| PARIS            | NANTES           | 385.0 |
| PARIS            | CLERMONT-FERRAND | 420.0 |
| PARIS            | LYON             | 470.0 |
| CLERMONT-FERRAND | MONTPELLIER      | 335.0 |
| CLERMONT-FERRAND | TOULOUSE         | 375.0 |
| LYON             | MONTPELLIER      | 305.0 |
| LYON             | MARSEILLE        | 320.0 |
| MONTPELLIER      | TOULOUSE         | 240.0 |
| MARSEILLE        | NICE             | 205.0 |
+------------------+------------------+-------+
```

Arbre modélisé

Figure 4-21 *Graphe orienté*



Constituer une liste des ascendants

La requête suivante parcourt le graphe récursivement pour extraire le nombre d'étapes des différents trajets entre les villes de Paris et Toulouse. La première CTE définit de manière unique le point de départ puis la construction des segments s'opère sur la base d'une jointure. La construction du chemin final nécessite de prévoir une taille maximale de la chaîne (ici 50 caractères).

```
WITH RECURSIVE graphe_oriente (ville_vers, etape, distance, trajet) AS
   (SELECT DISTINCT ville_de, 0, 0,
```

```
           CAST('PARIS' AS CHAR(50))
           FROM Autoroutes WHERE ville_de = 'PARIS'
    UNION  ALL
    SELECT a.ville_vers, g.etape + 1, g.distance+a.km,
           CAST(CONCAT(g.trajet,',',a.ville_vers) AS CHAR(50))
           FROM Autoroutes a INNER JOIN graphe_oriente g
           ON   g.ville_vers  = a.ville_de)
SELECT trajet, etape AS etapes, distance
FROM   graphe_oriente
WHERE  ville_vers = 'TOULOUSE'
ORDER BY distance;
```

```
+-----------------------------------------+--------+----------+
| trajet                                  | etapes | distance |
+-----------------------------------------+--------+----------+
| PARIS,CLERMONT-FERRAND,TOULOUSE         |      2 |      795 |
| PARIS,CLERMONT-FERRAND,MONTPELLIER,TOULOUSE |      3 |      995 |
| PARIS,LYON,MONTPELLIER,TOULOUSE         |      3 |     1015 |
+-----------------------------------------+--------+----------+
```

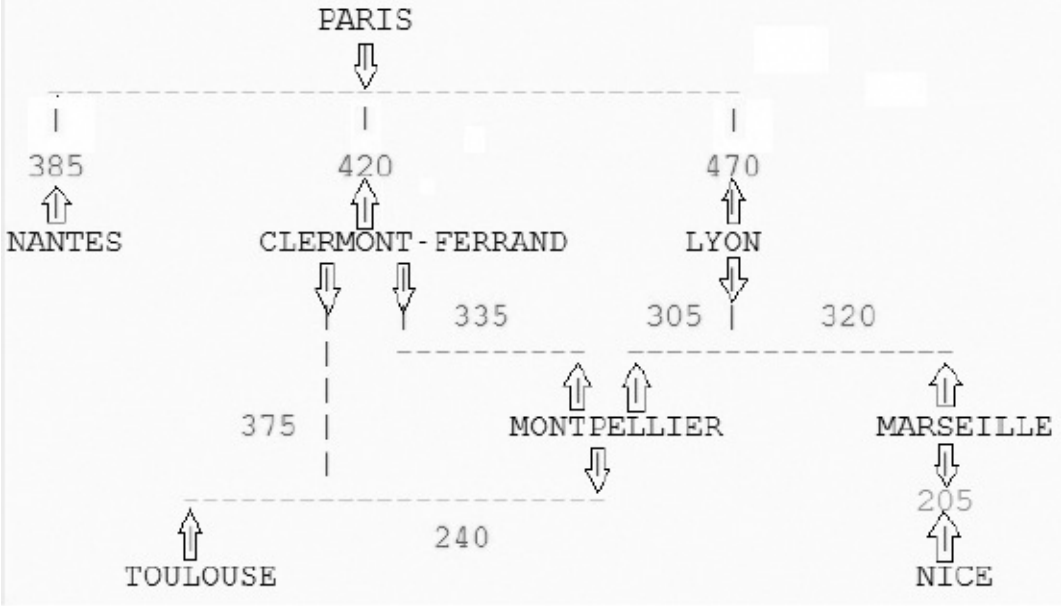
Attention à la présence d'éventuels cycles, un parcours sans fin s'opérerait alors et conduirait à une erreur système.

La section qui suit décrit le moyen d'éviter cette limitation.

Parcours d'un graphe non orienté

En insérant à chaque segment le segment inverse, on oriente le graphe qui contient désormais des cycles, notamment au niveau du triangle Clermont-Ferrand-Montpellier-Toulouse.

Figure 4-22 *Graphe non orienté*



Pour se débarrasser des cycles, vous devrez d'abord comparer tout chemin déjà parcouru avec le nœud courant. Ensuite, vous devez conditionner un nombre maximal d'itérations récursives.

La requête suivante teste d'une part si la ville d'arrivée (*ville_vers*) n'a pas été déjà empruntée dans les chemins construits. Cela s'opère à l'aide de la fonction INSTR, qui teste la présence d'une chaîne dans une autre. D'autre part, la recherche dans le graphe se limite ici à dix étapes. Une nouvelle route apparaît, puisque Montpellier est située sur le chemin de Clermont et de Lyon.

```
WITH RECURSIVE graphe_oriente (ville_vers, etape, distance, trajet) AS
   (SELECT DISTINCT ville_de, 0, 0,
           CAST('PARIS' AS CHAR(50))
           FROM Autoroutes WHERE ville_de = 'PARIS'
    UNION  ALL
    SELECT a.ville_vers, g.etape + 1, g.distance+a.km,
             CAST(CONCAT(g.trajet,',',a.ville_vers) AS CHAR(50))
           FROM Autoroutes a INNER JOIN graphe_oriente g
           ON    g.ville_vers  = a.ville_de
               WHERE INSTR(g.trajet,a.ville_vers) = 0
               AND   g.etape < 10)
SELECT trajet, etape AS etapes, distance
FROM   graphe_oriente
WHERE  ville_vers = 'TOULOUSE'
ORDER BY distance;
```

```
+-------------------------------------------------+--------+----------+
| trajet                                          | etapes | distance |
+-------------------------------------------------+--------+----------+
```

```
| PARIS,CLERMONT-FERRAND,TOULOUSE                 |      2 |      795 |
| PARIS,CLERMONT-FERRAND,MONTPELLIER,TOULOUSE     |      3 |      995 |
| PARIS,LYON,MONTPELLIER,TOULOUSE                 |      3 |     1015 |
| PARIS,LYON,MONTPELLIER,CLERMONT-FERRAND,TOULOUSE |      4 |     1485 |
+-------------------------------------------------+--------+----------+
```

Transformations de résultats

Résultats en HTML ou XML

Les options « `--html` » et « `--xml` » (étudiées dans l'introduction) permettent de formater le résultat des extractions aux standards du Web. Le format HTML produit une structure `TABLE`. Le format XML produit un document de racine `resultset` dont l'attribut `statement` contiendra la requête, puis, pour chaque ligne retournée, une balise `row` sera créée, contenant autant de balises `field` que nécessaire.

À titre d'exemple, la requête précédente (division exacte) produira les résultats suivants.

Tableau 4-57 Résultats formatés pour HTML ou XML

Connexion avec `--hmtl`	Connexion avec `--xml`

`<TABLE BORDER=1>`
`<TR><TH>immat</TH>`
`<TH>typeAv</TH></TR>`
`<TR><TD>A3</TD>`
`<TD>Mercure</TD></TR>`
`</TABLE>`
	`<?xml version="1.0"?>`
	`<resultset`
	`statement="SELECT DISTINCT immat, typeAv`
	`FROM Affretements aliasAff`
	`WHERE NOT EXISTS`
	`(SELECT comp FROM Compagnie WHERE pays = 'F'`
	`AND comp NOT IN`
	`(SELECT compa FROM Affretements`
	`WHERE immat = aliasAff.immat))`
	`AND NOT EXISTS`
	`(SELECT compa FROM Affretements`
	`WHERE immat = aliasAff.immat`
	`AND compa NOT IN`
	`(SELECT comp FROM Compagnie`
	`WHERE pays = 'F'))"`
	`xmlns:xsi="http://www.w3.org/2001/XMLSchemainstance">`
	`<row>`
	`<field name="immat">A3</field>`
	`<field name="typeAv">Mercure</field>`
	`</row>`
	`</resultset>`

Écriture dans un fichier

Mécanisme inverse à `LOAD DATA INFILE` étudié au chapitre 2, l'exportation de données (au format de fichiers) extraites à l'aide d'une requête peut être

programmée à l'aide de la directive INTO OUTFILE de l'instruction SELECT. Une telle requête écrit dans un fichier dans un répertoire du serveur. Le privilège FILE est requis. Le fichier cible doit être inexistant avant d'exécuter son chargement. La syntaxe simplifiée de cette directive est la suivante :

```
SELECT [ { DISTINCT | DISTINCTROW } | ALL ] listeColonnes
 FROM nomTable1 [,nomTable2]. [ WHERE condition ]
 INTO OUTFILE 'cheminEtNomFichier'
        [FIELDS [TERMINATED BY 'string']
                [[OPTIONALLY] ENCLOSED BY 'char']
                [ESCAPED BY 'char' ] ]
        [LINES   [STARTING BY 'string']
                [TERMINATED BY 'string'] ];
```

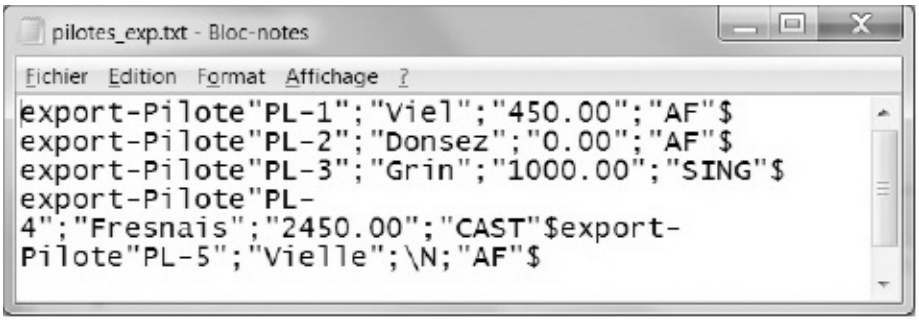
- FIELDS décrit comment seront formatées dans le fichier les colonnes extraites de(s) table(s). En l'absence de cette clause, TERMINATED BY vaut '\t', ENCLOSED BY vaut '' et ESCAPED BY vaut '\\'.
  - FIELDS TERMINATED BY décrit le caractère qui sépare deux valeurs de colonnes.
  - FIELDS ENCLOSED BY permet de contrôler le caractère qui encadrera chaque valeur de colonne.
  - FIELDS ESCAPED BY permet de contrôler les caractères spéciaux.
- LINES décrit comment seront écrites dans le fichier les lignes extraites de(s) table(s). En l'absence de cette clause, TERMINATED BY vaut '\n' et STARTING BY vaut ''.

Considérons la requête suivante qui exporte la totalité des enregistrements de la table Pilote en séparant les colonnes par le signe ;, encadrant les données par une double quote et terminant chaque ligne par le caractère $. Notez l'utilisation du double *back-slash* pour désigner une arborescence Windows, de \N pour indiquer le caractère NULL et \n pour le retour à la ligne (qui ne peut pas être opérant selon les versions de MySQL…).

```
SELECT *
    INTO OUTFILE 'C:\\Donnees\\dev\\pilotes_exp.txt'
    FIELDS TERMINATED BY ';' ENCLOSED BY '"'
    LINES STARTING BY 'export-Pilote' TERMINATED BY '$\n'
 FROM Pilote;
```

Le fichier « pilotes_exp.txt » créé sera situé dans le répertoire « C:\Donnees\dev ».

Figure 4-23 *Fichier créé dynamiquement par la requête*



Exercices

Les objectifs de ces exercices sont :

- de créer dynamiquement des tables et leurs données ;
- d'écrire des requêtes monotables et multitables ;
- de réaliser des modifications synchronisées ;
- de composer des jointures et des divisions.

Exercice 4.1

Création dynamique de tables

Écrire le script `créaDynamique.sql` permettant de créer les tables `Softs` et `PCSeuls` suivantes (en utilisant la directive `AS SELECT` de la commande `CREATE TABLE`). Vous ne poserez aucune contrainte sur ces tables. Penser à modifier le nom des colonnes.

Figure 4-24 *Structures des nouvelles tables*

Softs

nomSoft	version	prix

PCSeuls

nP	nomP	seq	ad	typeP	salle

La table `Softs` sera construite sur la base de tous les enregistrements de la table `Logiciel` que vous avez créée et alimentée précédemment. La table `PCSeuls` doit seulement contenir les enregistrements de la table `Poste`, qui sont de type 'PCWS' ou 'PCNT'. Vérifier :

```
SELECT * FROM Softs;
SELECT * FROM PCSeuls;
```

Exercice 4.2

Requêtes monotables

Écrire le script `requêtes.sql` permettant d'extraire, à l'aide d'instructions `SELECT`, les données suivantes :

Type du poste 'p8'.

Noms des logiciels 'UNIX'.

Noms, adresses IP, numéros de salle des postes de type 'UNIX' ou 'PCWS'.

Même requête pour les postes du segment '130.120.80' triés par numéros de salles décroissants.

Numéros des logiciels installés sur le poste 'p6'.

Numéros des postes qui hébergent le logiciel 'log1'.

Noms et adresses IP complètes (ex : '130.120.80.01') des postes de type 'TX' (utiliser la fonction de concaténation).

Exercice 4.3

Fonctions et groupements

Pour chaque poste, le nombre de logiciels installés (en utilisant la table `Installer`).

Pour chaque salle, le nombre de postes (à partir de la table `Poste`).

Pour chaque logiciel, le nombre d'installations sur des postes différents.

Moyenne des prix des logiciels 'UNIX'.

Plus récente date d'achat d'un logiciel.

Numéros des postes hébergeant 2 logiciels.

Nombre de postes hébergeant 2 logiciels (utiliser la requête précédente en faisant un `SELECT` dans la clause `FROM`).

Exercice 4.4

Requêtes multitables

Opérateurs ensemblistes

Types de postes non recensés dans le parc informatique (utiliser la table `Types`).

Types existant à la fois comme types de postes et de logiciels.

Types de postes de travail n'étant pas des types de logiciels.

Jointures procédurales

Adresses IP complètes des postes qui hébergent le logiciel 'log6'.

Adresses IP complètes des postes qui hébergent le logiciel de nom 'Oracle 8'.

Noms des segments possédant exactement trois postes de travail de type 'TX'.

Noms des salles ou l'on peut trouver au moins un poste hébergeant le logiciel 'Oracle 6'.

Nom du logiciel acheté le plus récent (utiliser la requête 12).

Jointures relationnelles

Écrire les requêtes 18, 19, 20, 21 avec des jointures de la forme relationnelle. Numéroter ces nouvelles requêtes de 23 à 26.

Installations (nom segment, nom salle, adresse IP complète, nom logiciel, date d'installation) triées par segment, salle et adresse IP.

Jointures SQL2

Écrire les requêtes 18, 19, 20, 21 avec des jointures SQL2 (`JOIN`, `NATURAL JOIN`, `JOIN USING`). Numéroter ces nouvelles requêtes de 28 à 31.

Exercice 4.5

Modifications synchronisées

Écrire le script `modifSynchronisées.sql` pour ajouter les lignes suivantes dans la table `Installer` :

Figure 4-25 *Lignes à ajouter*

Installer

nPoste	*nLog*	**numIns**	dateIns	delai
...	...	...	...	...
p2	log6	*séquence...*	SYSDATE()	NULL
p8	log1		SYSDATE()	NULL
p10	log1		SYSDATE()	NULL

Écrire les requêtes `UPDATE` synchronisées de la forme suivante :

```
UPDATE table1 alias1
  SET colonne = (SELECT COUNT(*)
                 FROM  table2 alias2
                 WHERE alias2.colonneA = alias1.colonneB...);
```

Pour mettre à jour automatiquement les colonnes rajoutées :

`nbSalle` dans la table `Segment` (nombre de salles traversées par le segment) ;

`nbPoste` dans la table `Segment` (nombre de postes du segment) ;

`nbInstall` dans la table `Logiciel` (nombre d'installations du logiciel) ;

`nbLog` dans la table `Poste` (nombre de logiciels installés par poste).

Vérifier le contenu des tables modifiées (`Segment`, `Logiciel` et `Poste`).

Exercice 4.6

Opérateurs existentiels

Rajouter au script `requêtes.sql`, les instructions `SELECT` pour extraire les données suivantes :

Sous-interrogation synchronisée

Noms des postes ayant au moins un logiciel commun au poste 'p6' (on doit trouver les postes p2, p8 et p10).

Divisions

Noms des postes ayant les mêmes logiciels que le poste 'p6' (les postes peuvent avoir plus de logiciels que 'p6'). On doit trouver les postes 'p2' et 'p8' (division inexacte).

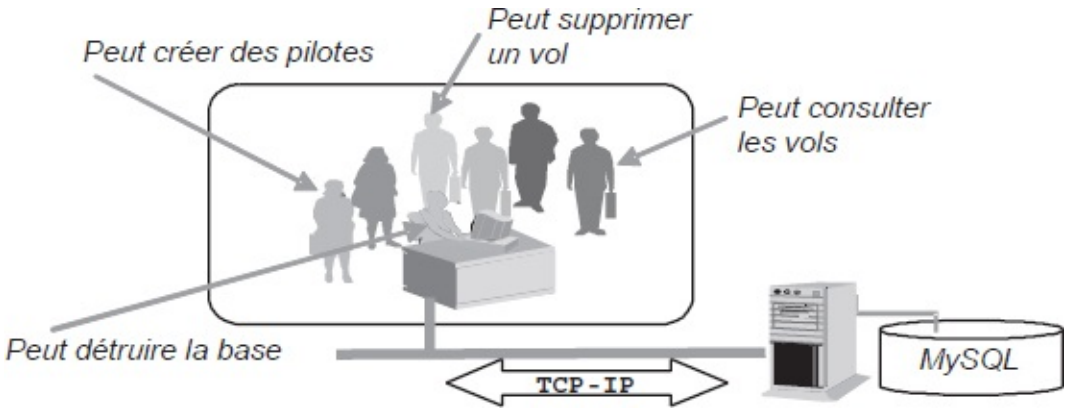
Noms des postes ayant exactement les mêmes logiciels que le poste 'p2' (division exacte), on doit trouver 'p8'.

Chapitre 5

Contrôle des données

Comme dans tout système multi-utilisateur, l'usager d'un SGBD doit être identifié avant de pouvoir utiliser des ressources. Les accès aux informations et à la base de données doivent être contrôlés à des fins de sécurité et de cohérence. La figure suivante illustre un groupe d'utilisateurs dans lequel existe une classification entre ceux qui peuvent consulter, mettre à jour, supprimer des enregistrements, voire les tables.

Figure 5-1 *Conséquences de l'aspect multi-utilisateur*



Nous verrons dans cette section les aspects du langage SQL qui concernent le contrôle des données et des accès. Nous étudierons :

- la gestion des utilisateurs qui manipuleront des bases de données dans lequelles se trouvent des objets tels que des tables, index, séquences (pour l'instant implémentées par des colonnes `AUTO_INCREMENT`), vues, procédures, etc. ;
- la gestion des privilèges qui permettent de donner des droits sur la base de données (privilèges système) et sur les données de la base (privilèges objet) ;
- la gestion des vues ;

- l'utilisation du dictionnaire des données (base de données `information_schema`).

Le chapitre 9 détaille quelques outils graphiques qui permettent de s'affranchir d'écrire des instructions SQL.

Gestion des utilisateurs

Présenté rapidement à l'introduction, nous verrons qu'un utilisateur (*user*) est identifié par MySQL par son nom et celui de la machine à partir de laquelle il se connecte. Cela fait, il pourra accéder à différents objets (tables, vues, séquences, index, procédures, etc.) d'une ou de plusieurs bases sous réserve d'avoir reçu un certain nombre de privilèges.

Classification

Les types d'utilisateurs, leurs fonctions et leur nombre peuvent varier d'une base à une autre. Néanmoins, pour chaque base de données en activité, on peut classifier les utilisateurs de la manière suivante :

- Le DBA (*DataBase Administrator*). Il en existe au moins un. Une petite base peut n'avoir qu'un seul administrateur. Une base importante peut en regrouper plusieurs qui se partagent les tâches suivantes :
  - installation et mises à jour de la base et des outils éventuels ;
  - gestion de l'espace disque et des espaces pour les données ;
  - gestion des utilisateurs et de leurs objets (s'ils ne les gèrent pas eux-mêmes) ;
  - optimisation des performances ;
  - sauvegardes, restaurations et archivages ;
  - contact avec le support technique.
- L'administrateur réseau (qui peut être le DBA) se charge de la configuration des couches client pour les accès distants.
- Les développeurs qui conçoivent et mettent à jour la base. Ils peuvent aussi agir sur leurs objets (création et modification des tables, index, séquences, etc.). Ils transmettent au DBA leurs demandes spécifiques (stockage, optimisation, sécurité).

- Les administrateurs d'application qui gèrent les données manipulées par la ou les applications. Pour les petites et les moyennes bases, le DBA joue ce rôle.
- Les utilisateurs qui se connectent et interagissent avec la base à travers les applications ou à l'aide d'outils (interrogations pour la génération de rapports, ajouts, modifications ou suppressions d'enregistrements).

Tous seront des utilisateurs (au sens MySQL) avec des privilèges différents.

Création d'un utilisateur [CREATE USER]

Pour pouvoir créer un utilisateur, vous devez posséder le privilège `CREATE USER` ou `INSERT` sur la base système `mysql` (car c'est la table `mysql.user` qui stockera l'existence de ce nouvel arrivant).

La syntaxe de création d'un utilisateur est la suivante :

```
CREATE USER utilisateur
     [IDENTIFIED BY [PASSWORD] 'motdePasse']
    [,utilisateur2 [IDENTIFIED BY [PASSWORD] 'motdePasse2'] ...];
```

- `IDENTIFIED BY 'motdePasse'` permet d'affecter un mot de passe (sensible à la casse) à un utilisateur (caractérisé par 16 caractères au maximum, sensibles aussi à la casse).

Le tableau suivant décrit la création d'un utilisateur (à exécuter en étant connecté en local en tant que `root`) :

Tableau 5-1 Création d'un utilisateur

Instruction SQL	Résultat
`CREATE USER soutou@localhost IDENTIFIED BY 'iut';`	*soutou* est déclaré « utilisateur à accès local », il devra se connecter à l'aide de son mot de passe.

Par défaut, les utilisateurs, une fois créés, n'ont aucun droit sur aucune base de données (à part en lecture écriture sur la base `test` et en lecture seule sur la base `information_schema`). La section « Privilèges » étudie ces droits.

Un utilisateur bien connu

Lors de l'installation, vous avez dû noter la présence de l'utilisateur `root` (mot de passe saisi à l'installation). Cet utilisateur est le DBA que MySQL vous offre. Il vous permettra d'effectuer vos tâches administratives en ligne de commande ou par une console graphique (créer des utilisateurs par exemple).

Liste des utilisateurs

Vous retrouverez les informations relatives à la connexion pour l'utilisateur `root` et les autres en interrogeant la table `user` de la base `mysql` (`mysql.user`).

L'extraction des colonnes `User` et `Host` renseigne les utilisateurs connus du serveur :

```
mysql> SELECT User,Host FROM mysql.user;
+--------+-----------+
| User   | Host      |
+--------+-----------+
| root   | localhost |
| soutou | localhost |
+--------+-----------+
```

En fonction de la version dont vous disposez, des options d'installation et des utilisateurs créés, vous n'obtiendrez pas un résultat similaire. La machine désignée par « `localhost` » spécifie que la connexion est autorisée en local. La même adresse peut être indiquée en IP V4 (`127.0.0.1`) et IP V6 (::1). L'absence de nom d'utilisateur signifie une connexion anonyme.

La machine désignée par « `%` » indique que la connexion est autorisée à partir de tout site, en supposant qu'un client MySQL est installé et qu'il est relié au serveur par TCP-IP (voir la section « `Table mysql.db` »).

Depuis la version 5.6, la table `user` dispose de la colonne `password_expired` qui précise les accès qui sont expirés et pour lesquels toute connexion est impossible jusqu'à l'établissement d'un nouveau mot de passe.

Modification d'un utilisateur

Le mot de passe d'un utilisateur peut être modifié indépendamment des

privilèges dont il dispose (que ce soit l'accès aux données aussi bien que des privilèges plus système comme la restriction du nombre de requêtes, les modifications ou les connexions à un serveur).

Pour changer un mot de passe jusqu'en version 5.7.5, utilisez la commande suivante :

```
SET PASSWORD FOR 'utilisateur'@'serveur' = PASSWORD('mot_de_passe');
```

Pour changer un mot de passe depuis la version 5.7.6, utilisez la commande suivante (qui vient de la syntaxe d'Oracle) :

```
ALTER USER 'utilisateur'@'serveur' IDENTIFIED BY 'mot_de_passe';
```

Une fois la modification d'un mot de passe réalisée, une connexion interdite se caractérise par l'erreur suivante : `ERROR 1045 (28000): Access denied for user 'utilisateur'@'serveur' (using password: xx)` (*xx* valant `YES` si un mot de passe est fournit, `NO` sinon). Par ailleurs, depuis la version 5.6, l'arrivée de l'instruction `ALTER USER` permet de faire expirer des mots de passe.

```
ALTER USER 'utilisateur'@'serveur' PASSWORD EXPIRE;
```

Renommer un utilisateur [RENAME USER]

Pour pouvoir renommer un utilisateur, vous devez posséder le privilège `CREATE USER` (ou le privilège `UPDATE` sur la base de données `mysql`). La syntaxe SQL est la suivante :

```
RENAME USER utilisateur TO nouveauNom;
```

Penser à spécifier l'accès complet à renommer (`user@machine`). Les privilèges et le mot de passe ne changent pas. Le tableau suivant décrit trois opérations de renommage d'utilisateurs (qui reviennent d'ailleurs à l'état initial).

Tableau 5-2 Renommer un utilisateur

Instruction SQL	Commentaire
`RENAME USER soutou@localhost TO christiansoutou@localhost;`	L'accès *soutou* en local est renommé *christiansoutou* en local.
	L'accès *christiansoutou* en

`RENAME USER christiansoutou@localhost TO christiansoutou@194.53.227.12;`	local est renommé *christiansoutou* en accès distant.
`RENAME USER christiansoutou@194.53.227.12 TO soutou@localhost`	L'accès est renommé complètement.

Verrouillage d'un utilisateur

Le mécanisme de verrouillage d'utilisateur est opérationnel depuis la version 5.7.6. Le directives ACCOUNT LOCK et ACCOUNT UNLOCK sont disponibles au sein des instructions CREATE USER et ALTER USER s'il s'agit d'un utilisateur existant. Par défaut un utilisateur est créé dans un état déverrouillé.

Les instructions suivantes permettent de créer un utilisateur local mais en lui interdisant toute connexion, de visualiser les caractéristiques de l'utilisateur, enfin de lui rendre l'accès à la base possible.

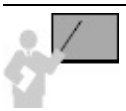

```
CREATE USER util_lock@localhost  IDENTIFIED BY 'util' ACCOUNT LOCK;
SHOW CREATE USER util_lock@localhost;
ALTER USER util_lock@localhost ACCOUNT UNLOCK;
```

Suppression d'un utilisateur [DROP USER]

Pour pouvoir supprimer un utilisateur, vous devez posséder le privilège CREATE USER (ou le privilège DELETE sur la base de données mysql). La syntaxe SQL est la suivante :

```
DROP USER utilisateur [,utilisateur2 ...];
```

Il faut spécifier l'accès à éliminer (user@machine). Tous les privilèges relatifs à cet accès sont détruits. Si l'utilisateur est connecté dans le même temps, sa suppression ne sera effective qu'à la fin de sa (dernière) session.



Aucune donnée d'aucune table que l'utilisateur aura mis à jour durant toutes ses connexions ne sera supprimée. Il n'y a pas de notion d'appartenance d'objets (tables, index, procédure, etc.) à un utilisateur. Tout ceci est relatif à la base de données (*database*).

Pour supprimer le compte `soutou` en local, la commande à lancer est :

```
DROP USER soutou@localhost;
```

Gestion des bases de données

Abordée brièvement à l'introduction, une base de données (*database*) regroupe essentiellement des tables sur lesquelles l'administrateur affectera des autorisations à des utilisateurs. Cette notion de *database* s'apparente plutôt à celle de *schéma* (connu des utilisateurs d'Oracle). D'ailleurs dans l'instruction de création les deux mots peuvent être utilisés.

Création d'une base [CREATE DATABASE]

Pour pouvoir créer une base de données, vous devez posséder le privilège `CREATE` sur la nouvelle base (ou au niveau global pour créer toute table).

```
CREATE {DATABASE | SCHEMA} [IF NOT EXISTS] nomBase
  [ [DEFAULT] CHARACTER SET nomJeu ]
      [ [DEFAULT] COLLATE nomCollation ];
```

- `IF NOT EXISTS` évite une erreur dans le cas où la base de données existe déjà (auquel cas elle ne sera pas remplacée).
- *`nomBase`* désigne le nom de la base (64 caractères maximum, caractères compris par le système de gestion de fichier du système d'exploitation, notamment respectant les règles de nommage des répertoires). Les caractères « / », « \ », ou « . » sont proscrits.
- `CHARACTER SET` indique le jeu de caractères associé aux données qui résideront dans les tables de la base.
- `COLLATE` définit la collation associée au jeu de caractères précédemment choisi.

Une fois créée, vous constaterez la présence d'un répertoire portant le nom de cette nouvelle base (par défaut dans `C:/ProgramData/MySQL/MySQL Server x.y/Data`

avec Windows 7). Ce répertoire contiendra les données de la nouvelle base.

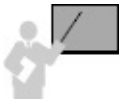
L'instruction suivante décrit la création d'une base de données dont le jeu de caractères est compatible avec le standard Unicode, en restant compatible avec la norme ASCII sensible aux accents et à la casse.

```
CREATE DATABASE bd_utf8
       DEFAULT CHARACTER SET utf8
       COLLATE utf8_bin;
```

Sélection d'une base de données [USE]

Ceux qui ont travaillé sous *Dbase* se souviennent de l'instruction USE qui désignait la table courante dans un programme. Pour MySQL, USE sélectionne une base de données qui devient active dans une session.

```
USE nomBase;
```



Si vous désirez travailler simultanément dans différentes bases de données, faites toujours préfixer le nom des tables par celui de la base par la notation pointée (*nomBase.nomTable*).

Le code suivant permet de créer une table dans la nouvelle base (ici *bd_utf8*) et d'y insérer une ligne tout en étant positionné dans une autre base de données (ici *bdutil*).

Figure 5-2 *Insertion dans la base non courante*

```
mysql> USE bdutil;
Database changed

mysql> CREATE TABLE bd_utf8.test_USE
    -> (col VARCHAR(6), col2 VARCHAR(30));
Query OK, 0 rows affected (0.08 sec)

mysql> INSERT INTO bd_utf8.test_USE VALUES ('RiM$', 'Réseaux et Télécoms');
Query OK, 1 row affected (0.01 sec)
```

Modification d'une base [ALTER DATABASE]

`ALTER DATABASE` vous permet de modifier le jeu de caractères par défaut d'une base de données. Pour pouvoir changer ainsi une base, vous devez avoir le privilège `ALTER` sur la base de données en question.

```
ALTER {DATABASE nomBase
  [ [DEFAULT] CHARACTER SET nomJeu ]
     [ [DEFAULT] COLLATE nomCollation ];
```

L'instruction suivante modifie la collation de la nouvelle base en la rendant insensible à la casse et à l'accentuation.

```
ALTER DATABASE bd_utf8 DEFAULT COLLATE utf8_general_ci;
```

Suppression d'une base [DROP DATABASE]

Pour pouvoir supprimer une base de données, vous devez posséder le privilège `DROP` sur la base (ou au niveau global pour effacer toute base). Cette commande détruit tous les objets (tables, index, etc.) et le répertoire contenus dans la base.

```
DROP {DATABASE | SCHEMA} [IF EXISTS] nomBase;
```

- `IF EXISTS` évite une erreur dans le cas où la base de données n'existerait pas.
- Cette instruction retourne le nombre de tables qui on été supprimées (fichiers à l'extension « `.frm` »).

L'instruction suivante supprime la base récemment créée.

```
DROP DATABASE bd_utf8;
```

Création des espaces de stockage

Afin de mieux gérer l'espace de stockage et la manière dont les tables et index sont organisés physiquement, la notion de *general tablespace* est apparue en version 5.7.6. Cette fonctionnalité permet de regrouper, dans une même entité logique, des tables de différents formats (*redundant*, *compact*, *compressed* et *dynamic*) au sein d'un fichier dont on choisit le nom et l'emplacement (qui peut être extérieur au répertoire par défaut des données). Pour les tables compressées, il faudra qu'elles disposent de la même taille de blocs.

La création d'un espace s'opère à l'aide de l'instruction suivante :


```
CREATE TABLESPACE nom_espace_logique
    ADD DATAFILE 'emplacement_et_nom_fichier.ibd'
    [FILE_BLOCK_SIZE = valeur]  [ENGINE [=] nom_moteur]
```

Une table peut être disposée dans un tel espace à sa création ou ultérieurement (avec ATER TABLE). Enfin, un espace logique peut être supprimé (avec DROP TABLESPACE) à la condition qu'il soit vidé de toute donnée. Le tableau suivant présente la syntaxe de ces actions.

Tableau 5-3 Gestion d'un espace logique

Instruction SQL	Résultat

`CREATE TABLESPACE gestion_vols`
`ADD DATAFILE 'C:/temp/clients/tsgescli.ibd'`
`ENGINE=INNODB;`
	Création de l'espace.

`CREATE DATABASE bd_billets;` `CREATE DATABASE bd_ventes;`	Création de deux bases.

`CREATE TABLE bd_billets.commandes`
`(id_cde NUMERIC(3), siege VARCHAR(3),`
`prix DECIMAL(4,2));`
`CREATE TABLE bd_ventes.reductions`
`(id_cde NUMERIC(3), siege VARCHAR(3),`
`reduc DECIMAL(4,2))`
`TABLESPACE=gestion_vols;`
	Création de deux tables dont une est disposée dans l'espace.

`ALTER TABLE bd_billets.commandes` `TABLESPACE=gestion_vols;`	Association de la deuxième table à l'espace.

`DROP TABLE bd_billets.commandes;`
`DROP TABLE bd_ventes.reductions;`
`DROP TABLESPACE gestion_vols;`
	Suppression de l'espace.

Privilèges

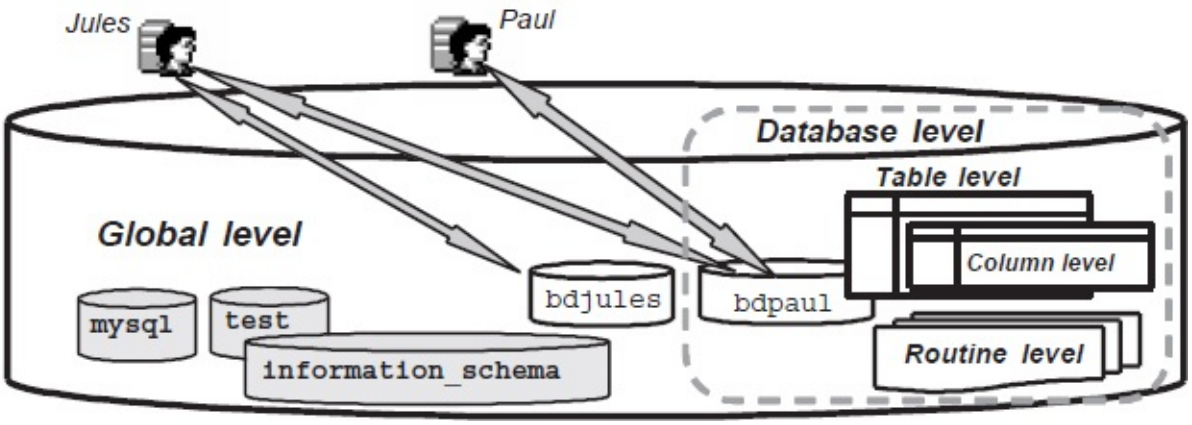
Depuis le début du livre, nous avons parlé de privilèges. Il est temps à présent de préciser ce que recouvre ce terme. Un privilège (sous-entendu *utilisateur*) est un droit d'exécuter une certaine instruction SQL (on parle de privilège *système*), ou un droit relatif aux données des tables situées dans différentes bases (on parle de privilège *objet*). La connexion, par exemple, sera considérée comme un privilège système bien que n'étant pas une commande SQL.

Les privilèges système diffèrent sensiblement d'un SGBD à un autre. Chez Oracle, il y en a plus d'une centaine, MySQL est plus modeste en n'en proposant qu'une vingtaine. En revanche, on retrouvera les mêmes privilèges objet (exemple : autorisation de modifier la colonne `nomComp` de la table `Compagnie`) qui sont attribués ou retirés par les instructions `GRANT` et `REVOKE`.

Niveaux de privilèges

La figure suivante illustre les différents niveaux de privilèges que l'on peut rencontrer :

Figure 5-3 *Niveaux de privilèges*



- *Global level* : privilèges s'appliquant à toutes les bases du serveur. Ces privilèges sont stockés dans la table `mysql.user` (exemple d'attribution d'un privilège global : `GRANT CREATE ON *.*`…).
- *Database level* : privilèges s'appliquant à tous les objets d'une base de données en particulier. Ces privilèges sont stockés dans les tables `mysql.db` et `mysql.host` (exemple d'attribution d'un privilège *database* : `GRANT SELECT ON bdpaul.*`…).
- *Table level* : privilèges s'appliquant à la globalité d'une table d'une base de données en particulier. Ces privilèges sont stockés dans la table `mysql.tables_priv` (exemple d'attribution d'un privilège *table* : `GRANT INSERT ON bdpaul.Avion`…).
- *Column level* : privilèges s'appliquant à une des colonnes d'une table d'une base de données en particulier. Ces privilèges sont stockés dans la table `mysql.columns_priv` (exemple d'attribution d'un privilège *column* :

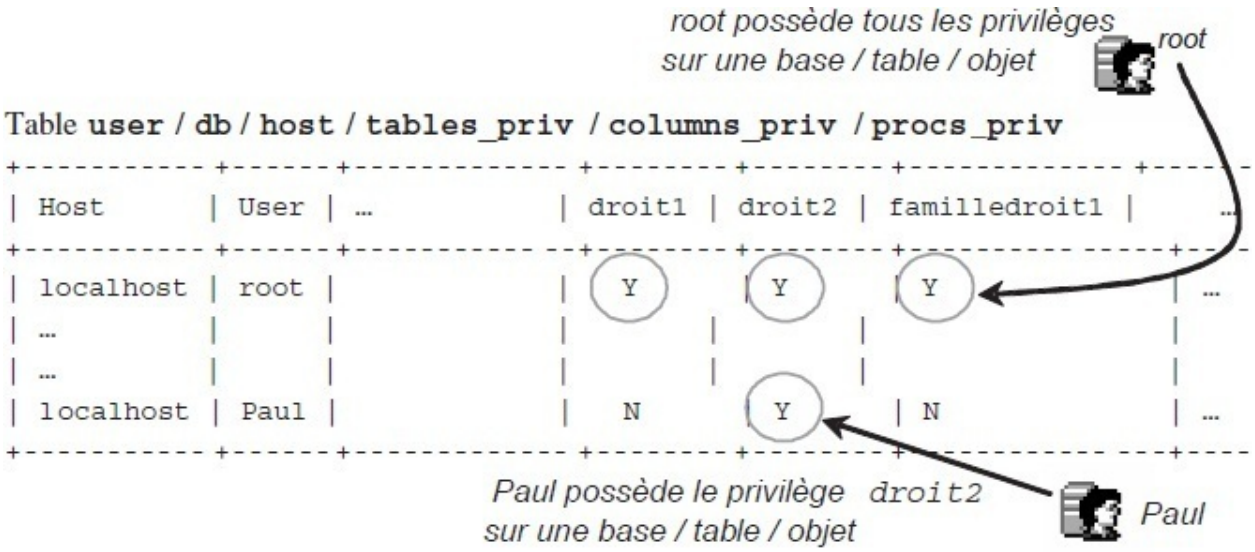
`GRANT UPDATE(nomComp)ON bdpaul.Compagnie`...).

- *Routine level* : privilèges globaux ou au niveau d'une base (`CREATE ROUTINE`, `ALTER ROUTINE`, `EXECUTE`, et `GRANT`) s'appliquant aux procédures cataloguées (étudiées au chapitre 7). Ces privilèges sont stockés dans la table `mysql.procs_priv` de la base `mysql` (exemple d'attribution d'un privilège *routine* : `GRANT EXECUTE ON PROCEDURE bdpaul.sp1`...).

Tables de la base `mysql`

Cinq tables de la base de données `mysql` suffisent à MySQL pour stocker les privilèges (système et objet) de tous les utilisateurs. La figure suivante illustre comment MySQL déduit toutes ces prérogatives toujours en fonction des accès (couple utilisateur, machine).

***Figure 5-4** Stockage des prérogatives*



La colonne `Db` est en plus dans les tables `host`, `tables_priv` et `columns_priv`, car elle est nécessaire pour désigner la base de données sur laquelle portera le droit ou la famille de droits.

Supposons, pour nos exemples, que l'utilisateur `Paul` (accès en local) et la base de données `bdpaul` soient créés.

```
CREATE DATABASE bdpaul;
CREATE USER Paul@localhost IDENTIFIED BY 'iut';
```

Table `mysql.user`

Présenté brièvement au début du chapitre. Cette table est composée d'une quarantaine de colonnes qui décrivent les privilèges au niveau global du serveur. Nous détaillons ici la signification des principales.

Privilèges objet (LMD) sur toutes les bases de données

La requête suivante extrait les prérogatives de `Paul` (et des autres). Pour l'instant, le caractère `'N'` étant dans toutes les colonnes, il ne peut ni interroger une table (`Select_priv`), ni insérer dans une table (`Insert_priv`), ni en modifier (`Update_priv`), ni en supprimer (`Delete_priv`), et ce quelle que soit la base de données (excepté les bases système `test` et `information_schema`) sur laquelle il voudra se connecter.

```
SELECT Host, User, Select_priv, Insert_priv, Update_priv, Delete_priv
       FROM mysql.user;
+-----------+------+-------------+-------------+-------------+-------------+
| Host      | User | Select_priv | Insert_priv | Update_priv | Delete_priv |
+-----------+------+-------------+-------------+-------------+-------------+
| localhost | root | Y           | Y           | Y           | Y           |
| localhost |      | Y           | Y           | Y           | Y           |
| %         |      | N           | N           | N           | N           |
| localhost | Paul | N           | N           | N           | N           |
+-----------+------+-------------+-------------+-------------+-------------+
```

Vous pouvez, par analogie, pour cet exemple et pour les suivants, découvrir les prérogatives des autres accès (ici `root` et *anonyme*).

Privilèges objet (LDD) sur toutes les bases de données

La requête suivante extrait les prérogatives à propos des instructions LDD. Pour l'instant, le caractère `'N'` étant dans toutes les colonnes, `Paul` ne peut ni créer une table ou une base (`Create_priv`), ni en supprimer (`Drop_priv`), ni créer ou supprimer un index (`Index_priv`), ni modifier la structure d'une table, la renommer ou modifier une base (`Alter_priv`), et ce quelle que soit la base de données (excepté les bases système `test` et `information_schema`).

```
SELECT Host, User, Create_priv, Drop_priv,Index_priv, Alter_priv
       FROM mysql.user;
+-----------+------+-------------+-----------+------------+------------+
| Host      | User | Create_priv | Drop_priv | Index_priv | Alter_priv |
+-----------+------+-------------+-----------+------------+------------+
| localhost | root | Y           | Y         | Y          | Y          |
| localhost |      | Y           | Y         | Y          | Y          |
| %         |      | N           | N         | N          | N          |
| localhost | Paul | N           | N         | N          | N          |
```

```
+-----------+------+-------------+-----------+------------+------------+
```

Privilèges système (LCD) sur toutes les bases de données

La requête suivante extrait les prérogatives à propos des instructions LCD. Pour l'instant, le caractère 'N' étant dans toutes les colonnes, `Paul` ne peut ni créer un utilisateur (`Create_user_priv`), ni transmettre des droits qu'il aura lui-même reçus (`Grant_priv`), ni lister les bases de données existantes (`Show_db_priv`), et ce quelle que soit la base de données.

```
SELECT Host,User, Create_user_priv, Grant_priv, Show_db_priv FROM mysql.user;
+-----------+------+-----------------+------------+--------------+
| Host      | User | Create_user_priv | Grant_priv | Show_db_priv |
+-----------+------+-----------------+------------+--------------+
| localhost | root | Y                | Y          | Y            |
| localhost |      | N                | Y          | Y            |
| %         |      | N                | N          | N            |
| localhost | Paul | N                | N          | N            |
+-----------+------+-----------------+------------+--------------+
```

Privilèges à propos des vues sur toutes les bases de données

La requête suivante extrait les prérogatives à propos des instructions relatives aux vues (*views* détaillées dans la section suivante). Pour l'instant, le caractère 'N' étant dans toutes les colonnes, `Paul` ne peut ni créer une vue (`Create_view_priv`), ni lister les vues existantes (`Show_view_priv`), et ce quelle que soit la base de données.

```
SELECT Host,User, Create_view_priv, Show_view_priv FROM mysql.user;
+-----------+------+-----------------+----------------+
| Host      | User | Create_view_priv | Show_view_priv |
+-----------+------+-----------------+----------------+
| localhost | root | Y                | Y              |
| localhost |      | N                | N              |
| %         |      | N                | N              |
| localhost | Paul | N                | N              |
+-----------+------+-----------------+----------------+
```

Privilèges à propos des procédures cataloguées sur toutes les bases de données

La requête suivante extrait les prérogatives à propos des procédures cataloguées (détaillées dans le chapitre 7). Pour l'instant, le caractère 'N' étant dans toutes les colonnes, `Paul` ne peut ni créer une procédure (`Create_routine_priv`), ni en modifier (`Alter_routine_priv`), ni en exécuter (`Execute_priv`), et ce quelle que soit la base de données.

```
SELECT Host,User, Create_routine_priv, Alter_routine_priv, Execute_priv
       FROM mysql.user;
+-----------+------+----------------------+--------------------+--------------+
| Host      | User | Create_routine_priv | Alter_routine_priv | Execute_priv |
+-----------+------+----------------------+--------------------+--------------+
| localhost | root | Y                    | Y                  | Y            |
| localhost |      | N                    | N                  | Y            |
| %         |      | N                    | N                  | N            |
| localhost | Paul | N                    | N                  | N            |
+-----------+------+----------------------+--------------------+--------------+
```

Privilèges à propos des restrictions d'utilisateur

La requête suivante extrait les prérogatives à propos des restrictions qu'on peut définir par accès. Pour l'instant, le chiffre étant à 0 dans toutes les colonnes, aucun accès (utilisateur) n'est limité concernant le nombre de requêtes (`max_questions`), de modifications (`max_updates`), de connexions par heure (`max_connections`) et de connexions simultanées (`max_user_connections`) à un serveur.

```
SELECT  Host, User, max_questions "Requetes", max_updates "Modifs" ,
        max_connections "Connexions", max_user_connections "Cx simult."
       FROM mysql.user;
+-----------+------+----------+--------+------------+------------+
| Host      | User | Requetes | Modifs | Connexions | Cx simult. |
+-----------+------+----------+--------+------------+------------+
| localhost | root |        0 |      0 |          0 |          0 |
| localhost |      |        0 |      0 |          0 |          0 |
| %         |      |        0 |      0 |          0 |          0 |
| localhost | Paul |        0 |      0 |          0 |          0 |
+-----------+------+----------+--------+------------+------------+
```

Privilèges non abordés

D'autres colonnes de la table `mysql.user` sont intéressantes, mais sortent un peu du cadre de ce livre. J'ai donc fait l'impasse sur `Create_tmp_table_priv` (sert à créer des tables temporaires), `Lock_tables_priv` (pose des verrous explicites), `Shutdown_priv` (arrête et redémarre le serveur), `Process_priv` et `Super_priv` (gèrent les processus), `File_priv` (accède aux fichiers du système d'exploitation), `Repl_slave_priv` et `Repl_client_priv` (utilisés pour des aspects de réplication de données).

Nul doute que vous saurez vous servir de ces privilèges en temps voulu, par analogie avec ceux que nous allons étudier.

Depuis la version 5.7.6, le privilège `REFERENCES` est opérationnel. Il permet de bénéficier de l'intégrité référentielle entre deux tables situées dans des bases différentes (dans l'exemple suivant, la table `Commande` de la base `bdjules` pourrait

faire référence (par une clé étrangère) à la table `Livre` située dans la base `bdpaul` sur la colonne clé primaire (ou unique).

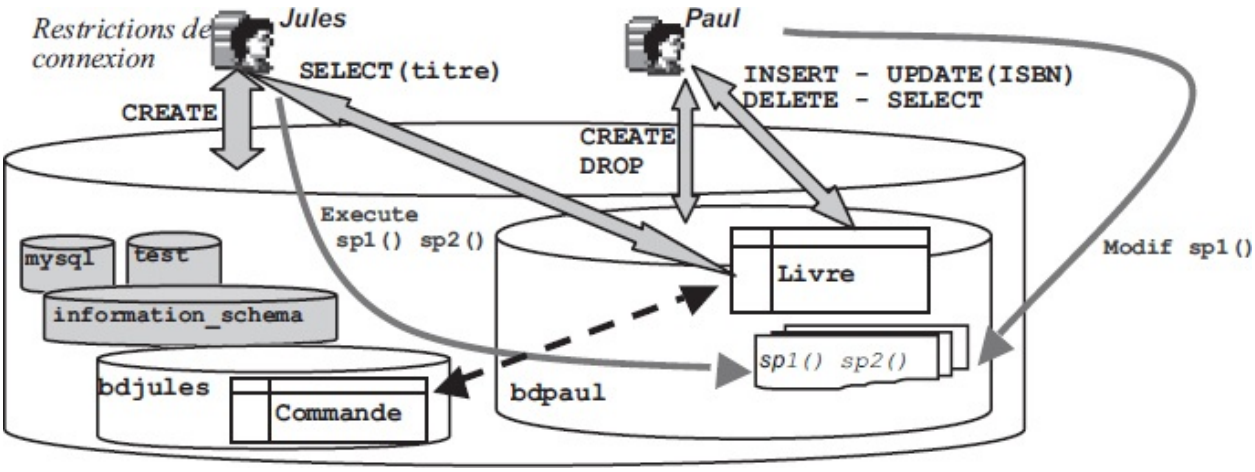
MySQL depuis 5.7

Avant de présenter les autres tables (`db`, `host`, `tables_priv`, `columns_priv` et `procs_priv`) de la base `mysql`, étudions les intructions relatives à l'attribution d'un privilège (`GRANT`), qu'il soit système ou objet, et celles relatives à la révocation d'un privilège (`REVOKE`).

Attribution de privilèges [GRANT]

La figure suivante illustre le contexte qui va servir d'exemple à l'attibution de prérogatives.

Figure 5-5 *Attribution de privilèges*



Syntaxe

L'instruction `GRANT` permet d'attribuer un (ou plusieurs) privilège(s) à propos d'un objet à un (ou plusieurs) bénéficiaire(s). L'utilisateur qui exécute cette commande doit avoir reçu lui-même le droit de transmettre ces privilèges (reçu avec la directive `GRANT OPTION`). Dans le cas de `root`, aucun problème, car il a implicitement tous les droits.

```
GRANT privilège [ (col1 [, col2...])] [,privilège2 ... ]
    ON [ {TABLE | FUNCTION | PROCEDURE} ]
       {nomTable | * | *.* | nomBase.*}
    TO utilisateur [IDENTIFIED BY [PASSWORD] 'password']
        [,utilisateur2 ...]
    [ WITH [ GRANT OPTION ]
           [ MAX_QUERIES_PER_HOUR nb ]
           [ MAX_UPDATES_PER_HOUR nb2 ]
           [ MAX_CONNECTIONS_PER_HOUR nb3 ]
           [ MAX_USER_CONNECTIONS nb4 ] ];
```

- *privilège* : description du privilège (ex : SELECT, DELETE, etc.), voir le tableau suivant.
- *col* précise la ou les colonnes sur lesquelles se portent les privilèges SELECT, INSERT ou UPDATE (exemple : UPDATE(typeAvion) pour n'autoriser que la modification de la colonne typeAvion).
- GRANT OPTION : permet de donner le droit de retransmettre les privilèges reçus à une tierce personne.

Le tableau suivant résume la signification des principaux privilèges à accorder ou à révoquer.

Tableau 5-4 Privilèges principaux pour GRANT et REVOKE

privilege	Commentaire
ALL [PRIVILEGES]	Tous les privilèges.
ALTER	Modification de base/table.
ALTER ROUTINE	Modification de procédure.
CREATE	Création de base/table.
CREATE ROLE	Création d'un rôle.
CREATE ROUTINE	Création de procédure.
CREATE USER	Création d'utilisateur.
CREATE VIEW	Création de vue.
DELETE	Suppression de données de table.
DROP	Suppression de base/table.
EXECUTE	Exécution de procédure.
INDEX	Création/Suppression d'index.
INSERT	Insertion de données de table.
REFERENCES	Clé étrangère distante.
SELECT	Extraction de données de table.

SHOW DATABASES	Lister les bases.
SHOW VIEW	Lister les vues d'une base.
SUPER	Exécution de commandes d'administration (CHANGE MASTER TO, KILL, PURGE BINARY LOGS, SET GLOBAL) et de débogage.
TRIGGER	Création de déclencheur.
UPDATE	Modification de données de table.
USAGE	Synonyme de « sans privilège », USAGE est utilisé pour conserver les privilèges précédemment définis tout en les restreignant avec des options.

Exemples

Le tableau suivant décrit l'affectation de quelques privilèges en donnant les explications associées.

Tableau 5-5 Affectation de privilèges

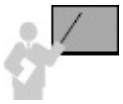
Instruction faite par root	**Explication**
`GRANT CREATE, DROP ON bdpaul.* TO 'Paul'@'localhost';`	Privilège système *database level* : Paul (en accès local) peut créer ou supprimer des tables dans la base bdpaul.
`GRANT INSERT, SELECT, DELETE, UPDATE(ISBN) ON bdpaul.Livre TO 'Paul'@'localhost';`	Privilège objet *table level* : Paul peut insérer, extraire, supprimer et modifier la colonne ISBN de la table Livre contenue dans la base bdpaul.
`GRANT ALTER ON bdpaul.Livre TO Paul'@'localhost' ;`	Privilège système *table level* : Paul peut modifier la structure ou les contraintes de la table Livre contenue dans la base bdpaul.
	Privilège objet *column level* :

`GRANT SELECT(titre)`
`ON bdpaul.Livre`
`TO 'Jules'@'localhost'`
`WITH GRANT OPTION;`
	`Jules` peut extraire seulement la colonne titre de la table `Livre` contenue dans la base `bdpaul`. Il pourra par la suite retransmettre éventuellement ce droit.

`GRANT CREATE ON *.*` `TO 'Jules'@'localhost';`	Privilège système *global level* : `Jules` peut créer des bases de données.

`GRANT USAGE ON bdpaul.*`
`TO 'Jules'@'localhost'`
`WITH`
`MAX_QUERIES_PER_HOUR 50`
`MAX_UPDATES_PER_HOUR 20`
`MAX_CONNECTIONS_PER_HOUR 6`
`MAX_USER_CONNECTIONS 3;`
	Privilège système *database level* :
	`Jules` ne peut lancer, chaque heure, que 50 `SELECT`, 20 `UPDATE`, se connecter 6 fois (dont 3 connexions simultanées) sur la base de données `bdpaul`.

`GRANT REFERENCES ON bdpaul.Livre` `TO 'Jules'@'localhost';`	Privilège objet *table level* : `Jules` peut bénéficier de la table de référence `Livre` située dans la base `bdpaul` pour s'assurer que le numéro du livre est bien existant, par exemple.



Tout ce que vous avez le droit de faire doit être explicitement autorisé par la commande `GRANT`. Ce qui n'est pas dit par `GRANT` n'est pas permis. Par exemple, `Jules` peut créer des bases, mais pas en détuire, `Paul` peut modifier le numéro ISBN d'un livre mais pas son titre, etc.

Voir les privilèges

La commande `SHOW GRANTS FOR` liste les différentes instructions `GRANT` équivalentes

à toutes les prérogatives d'un utilisateur donné. C'est bien utile quand vous avez attribué un certain nombre de privilèges à un utilisateur sans avoir pensé à les consigner dans un fichier de commande.

```
SHOW GRANTS FOR utilisateur;
```

Utilisons cette commande pour extraire les profils de `Jules`, de `Paul` et de l'administrateur en chef (accès en local). J'avoue avoir un peu retravaillé l'état de sortie (sans en modifier une ligne quand même).

```
SHOW GRANTS FOR 'Jules'@'localhost';
+-----------------------------------------------------------------------+
| Grants for Jules@localhost                                            |
+-----------------------------------------------------------------------+
| GRANT CREATE ON *.* TO 'Jules'@'localhost' IDENTIFIED BY PASSWORD
 '*6AE163FB9EE8BB011EB2E87316AA5BE563A6CDB7' WITH MAX_QUERIES_PER_HOUR 50
  MAX_UPDATES_PER_HOUR 20 MAX_CONNECTIONS_PER_HOUR 6 MAX_USER_CONNECTIONS 3 |
| GRANT SELECT (titre) ON `bdpaul`.`Livre` TO 'Jules'@'localhost'
      WITH GRANT OPTION                                                |
+-----------------------------------------------------------------------+
```

On remarque que MySQL a regroupé deux privilèges en une instruction `GRANT` (`CREATE` et les restrictions de connexions). Par là même, on se rend compte que les prérogatives de connexion sont au niveau *global*, bien qu'on les ait spécifiées au niveau *database*.

```
SHOW GRANTS FOR 'Paul'@'localhost';
+-----------------------------------------------------------------------+
| Grants for Paul@localhost                                             |
+-----------------------------------------------------------------------+
| GRANT USAGE ON *.* TO 'Paul'@'localhost' IDENTIFIED BY PASSWORD
      '*6AE163FB9EE8BB011EB2E87316AA5BE563A6CDB7'                      |
| GRANT CREATE, DROP ON `bdpaul`.* TO 'Paul'@'localhost'                |
| GRANT SELECT, INSERT, UPDATE (ISBN), DELETE, ALTER ON `bdpaul`.`Livre`
       TO 'Paul'@'localhost'                                           |
+-----------------------------------------------------------------------+
```

On remarque que MySQL a regroupé tous les privilèges sur la table `Livre` en une instruction `GRANT`. La première exprime le fait que `Paul` peut se connecter à toutes les bases (par `USE nomBase`), mais qu'il ne pourra travailler en réalité que dans `bdpaul`.

```
SHOW GRANTS FOR 'root'@'localhost';
+-----------------------------------------------------------------------+
| Grants for root@localhost                                             |
+-----------------------------------------------------------------------+
| GRANT ALL PRIVILEGES ON *.* TO 'root'@'localhost' IDENTIFIED BY PASSWORD
       '*387E25FE2CF7ED941E43A76AD9402825401698FC' WITH GRANT OPTION    |
+-----------------------------------------------------------------------+
```

On remarque que MySQL n'attribue qu'un seul droit, mais le plus fort ! Tous les droits (`ALL PRIVILEGES`) sur toutes les bases (`*.*`), avec en prime la clause `GRANT OPTION` qui permet de retransmettre n'importe quoi à n'importe qui, ou de tout révoquer.

Interrogeons à nouveau la table `user` de la base `mysql` stockant les prérogatives au niveau global du moment. Le droit de création en local de `Jules` apparaît sur toutes les bases.

```
SELECT Host,User, Create_priv,Drop_priv,Index_priv,Alter_priv FROM mysql.user;
+-----------+-------+-------------+-----------+------------+------------+
| Host      | User  | Create_priv | Drop_priv | Index_priv | Alter_priv |
+-----------+-------+-------------+-----------+------------+------------+
| …
| localhost | Paul  | N           | N         | N          | N          |
| localhost | Jules | Y           | N         | N          | N          |
+-----------+-------+-------------+-----------+------------+------------+
```

Les colonnes suivantes permettent de stocker les restrictions sur les connexions.

```
SELECT Host,User, max_questions "Requetes", max_updates "Modifs" ,
       max_connections "Connexions", max_user_connections "Cx simult."
       FROM mysql.user;
+-----------+-------+----------+--------+------------+------------+
| Host      | User  | Requetes | Modifs | Connexions | Cx simult. |
+-----------+-------+----------+--------+------------+------------+
| …
| localhost | Paul  |        0 |      0 |          0 |          0 |
| localhost | Jules |       50 |     20 |          6 |          3 |
+-----------+-------+----------+--------+------------+------------+
```

Analysons les autres tables de la base `mysql` pour découvrir les prérogatives des autres niveaux (*database*, *table*, *column* et *routine*).

Table `mysql.db`

La table `mysql.db` décrit les prérogatives au niveau *database*. Ainsi la colonne `Db` indique la base de données.

```
SELECT Host, User, Db, Create_priv, Drop_priv, Alter_priv FROM mysql.db;
+-----------+------+---------+-------------+-----------+------------+
| Host      | User | Db      | Create_priv | Drop_priv | Alter_priv |
+-----------+------+---------+-------------+-----------+------------+
| %         |      | test\_% | Y           | Y         | Y          |
| %         |      | test    | Y           | Y         | Y          |
| localhost | Paul | bdpaul  | Y           | Y         | N          |
+-----------+------+---------+-------------+-----------+------------+
```

Notez la possibilité de `Paul`, avec l'accès local, de créer et de supprimer des

tables dans la base `bdpaul`. Notez également la possibilité de créer, de supprimer, de modifier des tables par un accès distant anonyme sur la base `test`.

Table `mysql.host`

Cette table est étudiée à la section « Accès distants ».

Table `mysql.tables_priv`

La table `mysql.tables_priv` décrit les prérogatives objet au niveau *table*. Ainsi la colonne `Table_name` indique la table concernée, la colonne `Grantor` précise l'utilisateur ayant donné le droit. La colonne `Table_priv` est un `SET` contenant la liste des droits de l'utilisateur sur la table.

```
SELECT CONCAT(User,'@',Host) "Compte",
       CONCAT(Db,'.',Table_name) "Objet", Grantor, Table_priv
       FROM mysql.tables_priv;
+-----------------+--------------+---------------+---------------------------+
| Compte          | Objet        | Grantor       | Table_priv                |
+-----------------+--------------+---------------+---------------------------+
| Jules@localhost | bdpaul.Livre | root@localhost | Grant                     |
| Paul@localhost  | bdpaul.Livre | root@localhost | Select,Insert,Delete,Alter|
+-----------------+--------------+---------------+---------------------------+
```

On retrouve les quatre privilèges de `Paul` et celui de `Jules` (`GRANT OPTION` de `SELECT` sur la table).

Cette table possède aussi une colonne de nom `Timestamp` stockant l'instant au cours duquel s'est déroulée l'attribution (ou la révocation).

Table `mysql.columns_priv`

La table `mysql.columns_priv` décrit les prérogatives objet au niveau *column*. Ainsi la colonne `Table_name` indique la table concernée, la colonne `Column_name` précise la colonne concernée par le droit. La colonne `Column_priv` est un `SET` contenant la liste des droits de l'utilisateur sur la colonne de la table.

```
SELECT CONCAT(User,'@',Host) "Compte", CONCAT(Db,'.',Table_name) "Objet",
       Column_name, Column_priv FROM mysql.columns_priv;
+-----------------+--------------+-------------+-------------+
| Compte          | Objet        | Column_name | Column_priv |
+-----------------+--------------+-------------+-------------+
| Jules@localhost | bdpaul.Livre | titre       | Select      |
| Paul@localhost  | bdpaul.Livre | ISBN        | Update      |
+-----------------+--------------+-------------+-------------+
```

On retrouve le privilège de `Paul` et celui de `Jules` (portant ici sur la même table).

Table `mysql.procs_priv`

La table `mysql.procs_priv` décrit les prérogatives des procédures et des fonctions cataloguées au niveau *routine*.

Les privilèges `CREATE ROUTINE`, `ALTER ROUTINE`, `EXECUTE`, et `GRANT` s'appliquent sur les sous-programmes catalogués et peuvent être attribués au niveau *global* et *database*. `ALTER ROUTINE`, `EXECUTE`, et `GRANT` peuvent être assignés aussi au niveau *routine*.

En supposant que la base `bdpaul` contient la procédure cataloguée `sp1()` et la fonction `sp2()`, toutes deux écrites par `root`, le tableau suivant exprime l'affectation de quelques privilèges en donnant les explications associées.

Tableau 5-6 Affectation de privilèges

Instruction faite par `root`	**Explication**
`GRANT CREATE ROUTINE ON bdpaul.* TO 'Paul'@'localhost';`	Privilège système *database level* : `Paul` (en accès local) peut créer ou supprimer des sous-programmes catalogués dans la base `bdpaul`.
`GRANT ALTER ROUTINE ON PROCEDURE bdpaul.sp1 TO 'Paul'@'localhost';`	Privilège système *routine level* : `Paul` peut modifier la procédure `sp1` contenue dans la base `bdpaul`.
`GRANT EXECUTE ON PROCEDURE bdpaul.sp1 TO 'Jules'@'localhost';` `GRANT EXECUTE ON FUNCTION bdpaul.sp2 TO 'Jules'@'localhost';`	Privilèges système *routine level* : `Jules` peut exécuter la procédure `sp1` et la fonction `sp2` contenues dans la base `bdpaul`.

La colonne `Routine_name` de la table `mysql.procs_priv` désigne le nom du sous-programme catalogué. La colonne `Routine_type` précise le type du sous-programme catalogué (fonction ou procédure). La colonne `Grantor` indique l'utilisateur ayant compilé le sous-programme. La colonne `Proc_priv` est un `SET` contenant la liste des droits de l'utilisateur sur le sous-programme de la base.

Extrayons les privilèges relatifs aux sous-programmes au niveau *database*.

```
SELECT CONCAT(User,'@',Host) "Compte", Db,
    Create_routine_priv "create routine", Alter_routine_priv "alter routine",
    Execute_priv "exec. routine" FROM mysql.db;
+----------------+---------+---------------+---------------+--------------+
| Compte         | Db      | create routine | alter routine | exec. routine |
+----------------+---------+---------------+---------------+--------------+
| @%             | test\_% | N              | N             | N             |
| @%             | test    | N              | N             | N             |
| Paul@localhost | bdpaul  | Y              | N             | N             |
+----------------+---------+---------------+---------------+--------------+
```

On retrouve le privilège de `Paul`. Extrayons enfin les privilèges relatifs aux sous-programmes au niveau *routine*.

```
SELECT CONCAT(User,'@',Host) "Compte",
       CONCAT(,'.',Routine_name,':',Routine_type) "Objet",
       Grantor, Proc_priv FROM mysql.procs_priv;
+-----------------+----------------------+---------------+--------------+
| Compte          | Objet                | Grantor        | Proc_priv     |
+-----------------+----------------------+---------------+--------------+
| Jules@localhost | bdpaul.sp1:PROCEDURE | root@localhost | Execute       |
| Jules@localhost | bdpaul.sp2:FUNCTION  | root@localhost | Execute       |
| Paul@localhost  | bdpaul.sp1:PROCEDURE | root@localhost | Alter Routine |
+-----------------+----------------------+---------------+--------------+
```

On retrouve le privilège en modification de `sp1` pour `Paul`, et les deux privilèges d'exécution de `Jules`.

Révocation de privilèges [REVOKE]

La révocation d'un ou de plusieurs privilèges est réalisée par l'instruction `REVOKE`. Pour pouvoir révoquer un privilège, vous devez détenir (avoir reçu) au préalable ce même privilège avec l'option `WITH GRANT OPTION`.

Syntaxe

Dans la syntaxe suivante, les options sont les mêmes que pour la commande `GRANT`.

```
REVOKE privilège [ (col1 [, col2...])] [,privilège2 ... ]
    ON [ {TABLE | FUNCTION | PROCEDURE} ]
       {nomTable | * | *.* | nomBase.*}
    FROM utilisateur [,utilisateur2 ... ];
```

Exemples

Le tableau suivant décrit la révocation de certains privilèges acquis des utilisateurs `Paul` et `Jules`.

Tableau 5-7 Révocation de privilèges

Instruction faite par `root`	Explication
`REVOKE CREATE ON bdpaul.* FROM 'Paul'@'localhost';`	Privilège système *database level* : `Paul` (en accès local) ne peut plus créer de tables dans la base `bdpaul`.
`REVOKE ALTER,INSERT,UPDATE(ISBN) ON bdpaul.Livre FROM 'Paul'@'localhost';`	Privilège objet *table level* : `Paul` ne peut plus modifier la structure (ou les contraintes), insérer et modifier la colonne `ISBN` de la table `Livre` contenue dans la base `bdpaul`.
`GRANT USAGE ON bdpaul.* TO 'Jules'@'localhost' WITH MAX_QUERIES_PER_HOUR 0 MAX_UPDATES_PER_HOUR 0;`	Privilège système *database level* : `Jules` n'est plus limité en requêtes `SELECT` et `UPDATE` sur la base de données `bdpaul`. Ici c'est un `GRANT` qu'il faut faire, car il s'agit plus d'une restriction de connexion que d'une instruction SQL.

Vérifications

Une fois ces actualisations réalisées, les cinq tables de la base `mysql` contiennent un peu plus le caractère `'N'` qu'auparavant. Les colonnes `SET` des tables `mysql.tables_priv`, `mysql.columns_priv` et `mysql.procs_priv` sont également mises à jour. Ainsi, l'extraction du profil actuel de `Paul` au niveau *table* fait apparaître les deux seuls droits qu'il lui reste.

```
SELECT CONCAT(User,'@',Host) "Compte",CONCAT(Db,'.',Table_name)
"Objet",
       Grantor, Table_priv FROM mysql.tables_priv
       WHERE User='Paul' AND Host='localhost';
+---------------+-------------+---------------+--------------+
| Compte        | Objet       | Grantor       | Table_priv   |
+---------------+-------------+---------------+--------------+
| Paul@localhost | bdpaul.Livre | root@localhost | Select,Delete |
+---------------+-------------+---------------+--------------+
```

L'extraction du profil actuel de `Jules` au niveau *database* fait apparaître que les deux limitations de connexion sur les `SELECT` et `UPDATE` ont disparu.

```
SELECT Host,User, max_questions "Requetes", max_updates "Modifs" ,
       max_connections "Connexions", max_user_connections "Cx simult."
       FROM mysql.user WHERE User='Jules' AND Host='localhost';
```

```
+-----------+-------+----------+--------+------------+------------+
| Host      | User  | Requetes | Modifs | Connexions | Cx simult. |
+-----------+-------+----------+--------+------------+------------+
| localhost | Jules |        0 |      0 |          6 |          3 |
+-----------+-------+----------+--------+------------+------------+
```

Tout en une fois !

Il existe une instruction qui révoque tous les droits en une fois. Vous en avez assez d'un utilisateur qui ne cesse de vous casser les pieds, utilisez `REVOKE ALL PRIVILEGES`. Pensez quand même à sauvegarder au préalable le profil de `Jules` (`SHOW GRANT FOR`) pour pouvoir le faire travailler de nouveau quand vous serez calmé.

Selon la documentation officielle, la syntaxe suivante permet de supprimer toutes les prérogatives aux niveaux *global*, *database*, *table* et *column*. Et les privilèges *routine*, me direz-vous ? Ils ont dû l'oublier dans la documentation, mais ils sont aussi effacés, ne vous inquiétez pas, je l'ai testé.

```
REVOKE ALL PRIVILEGES, GRANT OPTION FROM utilisateur [, utilisateur2 ...] ;
```

Pour pouvoir annihiler ainsi un utilisateur, il faut détenir le privilège `CREATE USER` au niveau *global* ou le privilège `UPDATE` au niveau *database* sur la base `mysql`.

Ne confondez pas suppression de tous les droits d'un accès et suppression de l'utilisateur. Par analogie, les politiciens qui se voient retirer le droit de vote ne sont pas encore guillotinés (que je sache). Énervons-nous contre `Jules` :

```
REVOKE ALL PRIVILEGES, GRANT OPTION FROM 'Jules'@'localhost';
```

Attributions et révocations « sauvages »

Le caractère *open* de MySQL fait des fois bien les choses, mais, dans ce cas précis, je ne trouve pas. Ici mon billet d'humeur conteste la possibilité qui est donnée de modifier les cinq tables de la base `mysql` pour affecter tantôt un '`N`' par-ci, tantôt un '`Y`' par-là, ou encore pour mettre à jour un `SET` contenant `SELECT` par exemple, etc. Bref, un `UPDATE` raté, un `INSERT` dans la mauvaise colonne, un `DELETE` sans `WHERE`, et vous mettez une panique noire (comme la couleur par défaut de l'interface de commande) dans vos bases. Vous pouvez vous-même empêcher toute connexion (même celle du `root`).

Sous Oracle, les commandes `GRANT` et `REVOKE` mettent à jour des tables système que vous pouvez interroger, mais que vous ne pouvez pas modifier. C'est

heureux.

En conclusion, je ne décrirai aucune de ces manipulations, d'abord parce que je n'ai pas envie de me tromper en faisant des tests et bouleverser ainsi inutilement ma configuration. Ensuite parce si vous voulez « bidouiller », allez consulter des sites Web ou d'autres ouvrages qui recopient la documentation sans quelquefois changer ni tester les exemples, vous m'en direz des nouvelles.

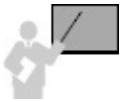
Vous voulez donner des droits : utilisez `GRANT` ; les reprendre : utilisez `REVOKE`, car :

- Ils sont programmés précisément pour ça.
- Les deux instructions sont dans la norme SQL.

Ne pas les employer, c'est comme acheter une quatre cylindres chez BMW (le motoriste est spécialiste des six en ligne) et verser en cachette du colza dans le réservoir chez l'agriculteur du coin en croyant économiser.

La seule utilisation acceptable, parce qu'on n'a pas le choix de faire autrement, concerne la mise à jour de la table `mysql.host` (décrite dans la section suivante). À configuration avancée, programmeur averti.

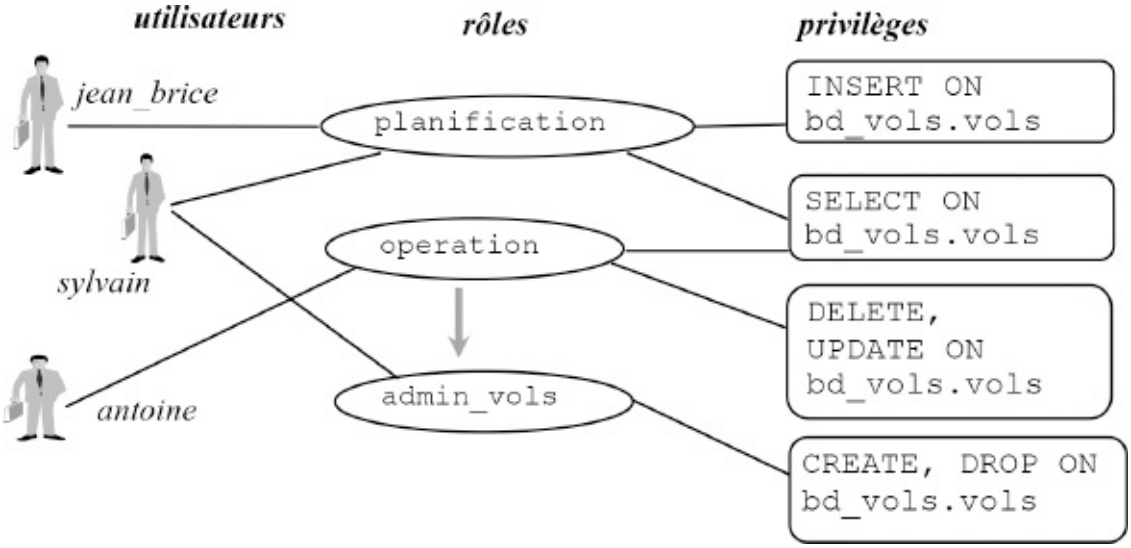
Rôles



Un rôle *(role)* est un ensemble nommé de privilèges (système ou objets). Un rôle est accordé à un ou plusieurs utilisateurs ou à un autre rôle. Ce mécanisme facilite la gestion des privilèges.

En supposant l'existence d'une base *bd_vols*, les rôles suivants vont permettre d'associer facilement un ensemble de privilèges à des utilisateurs qui sont susceptibles d'être remplacés dans le temps. Ainsi, les utilisateurs *jean_brice* et *sylvain* peuvent insérer et interroger la tables des vols mais seul *sylvain* pourra créer des tables dans la base. L'utilisateur *antoine* est le seul à pouvoir modifier ou supprimer des lignes de la table et à pouvoir créer des tables (le rôle *operation* hérite ici du rôle *admin_vols*).

***Figure 5-6** Attribution de privilèges via des rôles*



La chronologie des actions à entreprendre pour travailler avec des rôles est la suivante :

- créer le rôle (avec CREATE ROLE) ;
- alimenter le rôle avec des privilèges système ou objets (avec GRANT) ;
- attribuer le rôle à d'autres rôles ou à des utilisateurs (avec GRANT) ;
- faire évoluer le rôle dans le temps en ajoutant ou en enlevant des privilèges (avec GRANT ou REVOKE).

Création d'un rôle (CREATE ROLE)

Pour pouvoir créer un rôle vous devez posséder le privilège global CREATE ROLE ou CREATE USER. La syntaxe SQL est la suivante :

```
CREATE ROLE [IF NOT EXISTS] nom_role [,nom_role ...];
```

Le tableau suivant décrit la chronologie à respecter pour créer, alimenter et affecter ces rôles.

Tableau 5-8 Création et alimentation de rôles

Connexion administateur	Commentaire
`CREATE ROLE 'planification', 'operation',` `'admin_vols';`	Création des trois rôles.

`GRANT INSERT ON bd_vols.vols TO 'planification';`
`GRANT SELECT ON bd_vols.vols TO 'planification';`
`GRANT SELECT ON bd_vols.vols TO 'operation';`
`GRANT UPDATE,`
`DELETE ON bd_vols.vols TO 'operation';`
	Alimentation des rôles par des privilèges.

`GRANT CREATE,`
`DROP ON bd_vols.* TO 'admin_vols';`
`GRANT 'admin_vols' TO 'operation';`
	Alimentation d'un rôle par un autre rôle.

`GRANT 'planification' TO 'jean_brice'@'localhost',`
`'sylvain'@'localhost';`
`GRANT 'operation'  TO 'antoine'@'localhost';`
`GRANT 'admin_vols'  TO 'sylvain'@'localhost' ;`
	Affectation des rôles à des utilisateurs.

Visualisation d'un rôle

Pour vérifier la constitution d'un rôle, vous devrez utiliser la commande `SHOW GRANTS FOR` en précisant éventuellement le nom des rôles avec `USING` pour avoir le détail de chaque privilège.

Tableau 5-9 Vérification de rôles

Connexion administateur	Commentaire

`SHOW GRANTS FOR`
`'sylvain'@'localhost' ;`
	`+----------------------------------`
	`--`
	`| Grants for sylvain@localhost`
	`+----------------------------------`
	`--`
	` GRANT USAGE ON *.* TO`
	`` `sylvain`@`localhost` ``
	` GRANT`
	`` `admin_vols`@`%`,`planification`@`%` ``
	`TO`
	`` `sylvain`@`localhost` ``
	`+----------------------------------`
	`--`

	`+----------------------------------`
	`--`
	`| Grants for`
	`sylvain@localhost`
	`+----------------------------------`
	`--`

```
SHOW GRANTS FOR
  'sylvain'@'localhost'
 USING 'planification',
  'admin_vols';
```

```
 GRANT USAGE ON *.* TO
`sylvain`@`localhost`
 GRANT CREATE, DROP ON `bd_vols`.*
TO
  `sylvain`@`localhost`
 GRANT SELECT, INSERT ON
`bd_vols`.`vols` TO
  `sylvain`@`localhost`
 GRANT
`admin_vols`@`%`,`planification`@`%`
TO
  `sylvain`@`localhost`
+----------------------------------
---
```

Validation d'un rôle

Affecter un rôle à un utilisateur ne suffit pas à rendre effectif ce rôle pour les sessions à venir. En effet, dès la prochaine connexion de l'utilisateur *sylvain*, la fonction CURRENT_ROLE() retournera NONE pour préciser que l'utilisateur n'est associé à aucun rôle.

Pour assigner effectivement un ou plusieurs rôles à un utilisateur, l'administrateur doit utiliser la commande SET DEFAULT ROLE. L'option NONE annule les assignations précédentes. L'option ALL assigne tous les rôles. Il est aussi possible d'assigner des rôles choisis pour la session à suivre.

> **SET DEFAULT ROLE** {NONE | ALL | *nom_role* [,*nom_role ...*]} TO *user* [, *user ...*];

Tableau 5-10 Rendre effectif des rôles

Connexion	Instruction	Résultat
(root)	**SET DEFAULT ROLE** ALL TO 'jean_brice'@'localhost', 'sylvain'@'localhost', 'antoine'@'localhost';	Query OK, 0 rows affected (0.01 sec)
(sylvain)	SELECT **CURRENT_ROLE**();	+------------------------------------+ | CURRENT_ROLE() | +------------------------------------+ | `admin_vols`@`%`,`planification`@`%` | +------------------------------------+ 1 row in set (0.00 sec)

Si des privilèges sont ajoutés ou supprimés à un rôle qui est assigné, la session de l'utilisateur est impactée automatiquement pour être en phase avec la nouvelles constitution du rôle.

Modification et suppression d'un rôle

Modifier un rôle consiste à lui ajouter ou supprimer des privilèges. Les effets sont immédiats sur les sessions en cours qui concernent des utilisateurs affectés.

Tableau 5-11 Modification d'un rôle

Connexion administrateur	Conséquence
`mysql> GRANT INDEX ON bd_vols.* TO 'admin_vols';`	Le rôle *admin_vols* inclut désormais aussi le privilège de la création d'index.
`mysql> REVOKE DELETE ON bd_vols.vols FROM 'operation';`	Le rôle *operation* n'inclut plus désormais le privilège de suppression dans la table des vols.

La suppression d'un rôle s'opère à l'aide de l'instruction `DROP ROLE` qui est invocable, à condition de détenir le privilège `DROP ROLE` ou `CREATE USER`. Les effets sont immédiats sur toutes les sessions des utilisateurs concernés.

```
DROP ROLE [IF EXISTS] nom_role [, nom_role ...];
```

Ainsi, l'instruction suivante impacte les trois utilisateurs qui se voient démunis de tout privilège (à supposer qu'ils en aient détenu avant).

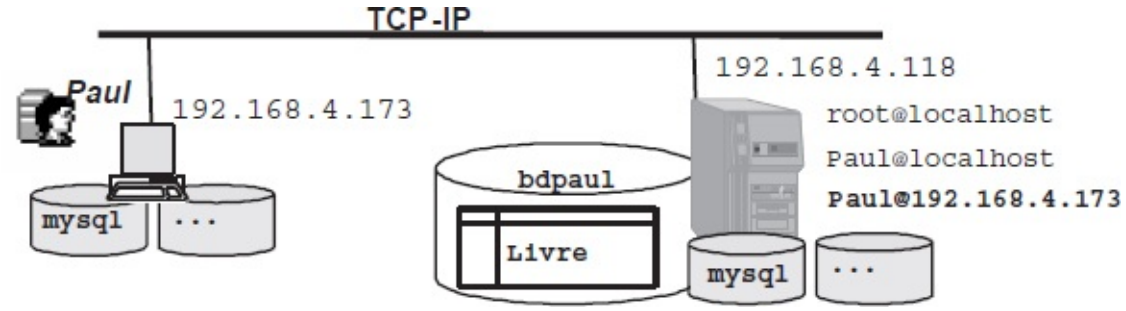
```
DROP ROLE 'admin_vols', 'operation', 'planification' ;
```

Accès distants

La figure suivante illustre la configuration de mon test. Un client est en 192.168.4.173 sur lequel sont installées les couches MySQL (*Complete Package* ou *Essentials Package*). Un serveur est en 192.168.4.118 équipé de MySQL *Complete Package*. Sur le serveur, `root` crée un accès à `Paul`, en précisant l'adresse de la machine client, et lui attribue un droit d'extraction de la table `Livre` sur la base `bdPaul`.

```
CREATE USER Paul@192.168.4.173 IDENTIFIED BY 'pauldistant';
GRANT SELECT ON bdpaul.Livre TO 'Paul'@'192.168.4.173';
```

***Figure 5-7** Accès distant par l'interface de commande MySQL*



Connexion par l'interface de commande

Sur le client, `Paul` se connecte au serveur dans une fenêtre de commande, en précisant l'adresse de la machine serveur, puis donne son mot de passe distant. Pensez à enlever les pare-feu Windows sur le client et le serveur (bloquant le port 3306).

```
mysql -h 192.168.4.118 -u Paul -p
```

`Paul` peut, à présent seulement, interroger la table distante, comme le montre la copie d'écran suivante :

Figure 5-8 *Interrogation distante par l'interface de commande MySQL*

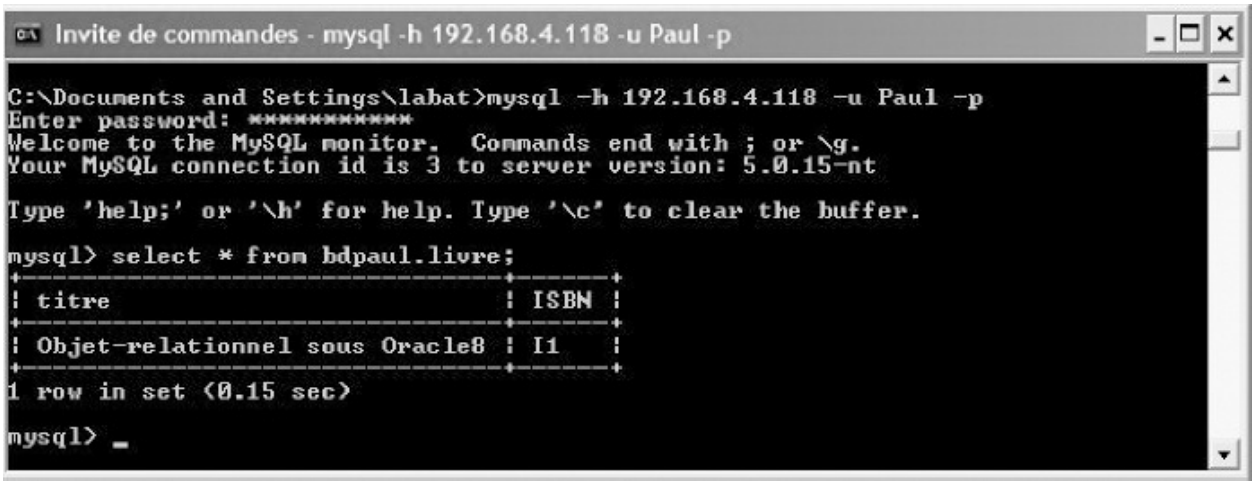


Table `mysql.host`

La table `mysql.host` est utilisée conjointement avec `mysql.db` et concerne les accès distants (plusieurs machines). Cette table n'est employée que pour les prérogatives au niveau *database*, indépendamment des utilisateurs. La structure

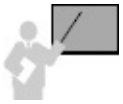
est la même que celle de mysql.db, à l'exception de la colonne User qui n'est pas présente. Le couple de colonnes (Host, Db) est unique.

Tableau 5-12 Tables pour les accès distants

Caractère	Signification pour mysql.db		Signification pour mysql.host	
	colonne Host	colonne Db	colonne Host	colonne Db
%	toute machine	toute base	toute machine	toute base
' ' (chaîne vide)	consultez la table mysql.host	toute base	toute machine	toute base

MySQL lit et trie les tables db (sur les colonnes Host, Db et User) et host (sur les colonnes Host et Db) en même temps qu'il parcourt la table user. Pour les opérations relatives aux bases (INSERT, UPDATE, etc.), MySQL interroge la table user. Si l'accès n'y est pas décrit, la recherche se poursuit dans les tables db et host. Si la colonne Host de la table db est renseignée en fonction de l'accès, l'utilisateur reçoit ses privilèges.

Si la colonne Host de la table db n'est pas renseignée (' '), cela signifie que la table host énumère les machines qui sont autorisées à accéder à une base de données en particulier. Si la machine ne correspond pas, l'accès n'est pas permis. Dans le cas contraire, les privilèges sont valués à 'Y' à partir d'une intersection (et pas d'une union) entre les tables db et host sur le couple (Host, Db).



La table mysql.host n'est mise à jour ni par GRANT, ni par REVOKE. Il faudra directement insérer (par INSERT), modifier (par UPDATE) ou supprimer (par DELETE) les lignes de cette table. Elle n'est pas utilisée par la plupart des serveur MySQL, car elle est dédiée à des usages très spécifiques (pour gérer un ensemble de machines à accès sécurisé, par exemple). Elle peut aussi être utilisée pour définir un ensemble de machines à accès non sécurisé.

En supposant que vous déclariez une machine à accès non sécurisé : camparols.gtr.fr. Il est possible d'autoriser l'accès sécurisé à toutes les autres

machines du réseau local. Ceci en ajoutant des enregistrements par `INSERT` dans la table `mysql.host` comme suit :

```
+-----------------+----+---------------------------
| Host            | Db | ...
+-----------------+----+---------------------------
| camparols.gtr.fr | %  | (tous les privilèges à 'N')
| %.gtr.fr         | %  | (tous les privilèges à 'Y')
+-----------------+----+---------------------------
```

Vous déclareriez l'inverse des conditions initiales en remplaçant les '`N`' par des '`Y`', et réciproquement. Dans tous les cas, il sera nécessaire de mettre à jour les autres tables pour affiner les privilèges.

Vues

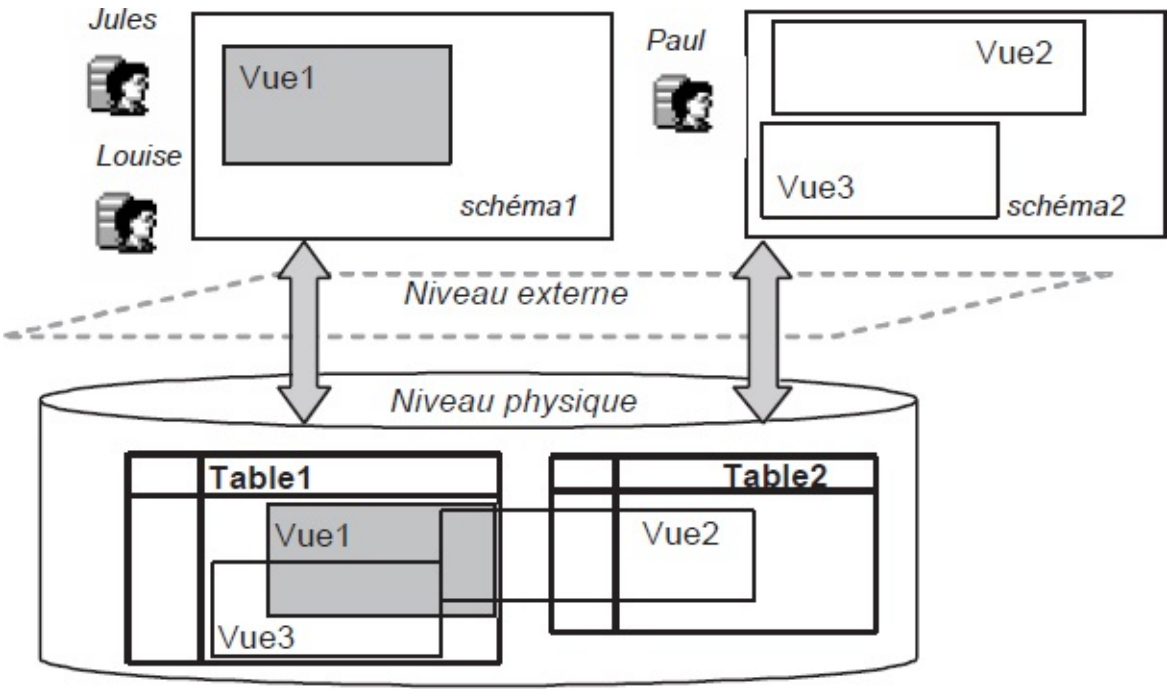
Outre le contrôle de l'accès aux données (privilèges), la confidentialité des informations est un aspect important qu'un SGBD relationnel doit prendre en compte. Depuis la version 5 de MySQL, la confidentialité est renforcée par l'utilisation de vues *(views)* qui agissent comme des fenêtres sur la base de données. Cette section décrit les différents types de vues qu'on peut rencontrer.

Les vues correspondent à ce qu'on appelle « le niveau externe » qui reflète la partie visible de la base de données pour chaque utilisateur.

Seules les tables contiennent des données et, pourtant, pour l'utilisateur, une vue apparaît comme une table. En théorie, les utilisateurs ne devraient accéder aux informations qu'en questionnant des vues. Ces dernières masquant la structure des tables interrogées. En pratique, la plupart des applications se passent de ce concept en manipulant directement les tables.

La figure suivante illustre ce qui a été dit en présentant trois utilisateurs. Ils travaillent chacun sur une base de données contenant des vues formées à partir de différentes tables.

Figure 5-9 *Les vues*



Une vue est considérée comme une table virtuelle car elle n'a pas d'existence propre. Seule sa structure est stockée dans le dictionnaire. Ses données seront extraites de la mémoire à partir des tables source, à la demande.

Une vue est créée à l'aide d'une instruction SELECT appelée « requête de définition ». Cette requête interroge une (ou plusieurs) table(s) ou vue(s). Une vue se recharge chaque fois qu'elle est interrogée.

Outre le fait d'assurer la confidentialité des informations, une vue est capable de réaliser des contrôles de contraintes d'intégrité et de simplifier la formulation de requêtes complexes. Dans certains cas, la définition d'une vue temporaire est nécessaire pour écrire une requête qu'il ne serait pas possible de construire à partir des tables seules. Attribuées comme des privilèges (GRANT), les vues améliorent la sécurité des informations stockées.

Création d'une vue [CREATE VIEW]

Pour pouvoir créer une vue dans une base, vous devez posséder le privilège CREATE VIEW et les privilèges en SELECT des tables présentes dans la requête de

définition de la vue. La syntaxe SQL de création d'une vue est la suivante :

```
CREATE [OR REPLACE] [ALGORITHM = {UNDEFINED | MERGE | TEMPTABLE}]
    VIEW [nomBase.]nomVue [(listecolonnes)]
    AS requêteSELECT
    [WITH [CASCADED | LOCAL] CHECK OPTION];
```

- `OR REPLACE` remplace la vue par la nouvelle définition, même si elle existait déjà (évite de détruire la vue avant de la recréer). Vous devez avoir le privilège `DELETE` sur la base pour bénéficier de cette directive.
- `ALGORITHM=MERGE` : la définition de la vue et sa requête sont fusionnées en interne.
- `ALGORITHM=TEMPTABLE` : les résultats sont extraits dans une table temporaire (`TEMPORARY`) qui est utilisée par la suite. Intéressant si les tables source sont sujettes à de nombreux verrous qui ne gênent plus la manipulation de la vue utilisant, elle, une table temporaire.
- Aucune option ou `ALGORITHM=UNDEFINED` : MySQL choisit la politique à adopter, souvent en faveur de `MERGE`, la seule qui convient aux vues modifiables.
- *`nomBase`* désigne le nom de la base de données qui hébergera la vue. En l'absence de ce paramètre, la vue est créée dans la base en cours d'utilisation.
- *`requêteSELECT`* : requête de définition interrogeant une (ou des) table(s) ou vue(s) pour charger les données dans la vue.

- La requête de définition ne peut interroger une table temporaire, ni contenir de paramètres ou de variables de session.
- Si la requête de définition sélectionne toutes les colonnes d'un objet source (`SELECT * FROM`…), et si des colonnes sont ajoutées par la suite à cet objet, la vue ne contiendra pas ces colonnes définies ultérieurement à elle. Il faudra recréer la vue pour prendre en compte l'évolution structurelle de l'objet source.

- `WITH CHECK OPTION` garantit que toute mise à jour de la vue par `INSERT` ou `UPDATE` s'effectuera conformément au prédicat contenu dans la requête de définition.

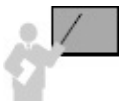
- Les paramètres LOCAL et CASCADED (par défaut) determinent la portée de la vérification quand une vue est définie à partir d'une autre vue. LOCAL restreint la vérification du prédicat à la vue elle-même. CASCADED permet d'étendre éventuellement les vérifications aux autres vues source de la vue qui vient d'être définie.

Classification

On distingue les vues simples des vues complexes en fonction de la nature de la requête de définition. Le tableau suivant résume ce que nous allons détailler au cours de cette section :

Tableau 5-13 Classification des vues

Requête de définition	Vue simple	Vue complexe
Nombre de tables	1	1 ou plusieurs
Fonction	non	oui
Regroupement	non	oui
Mises à jour possibles ?	oui	pas toujours

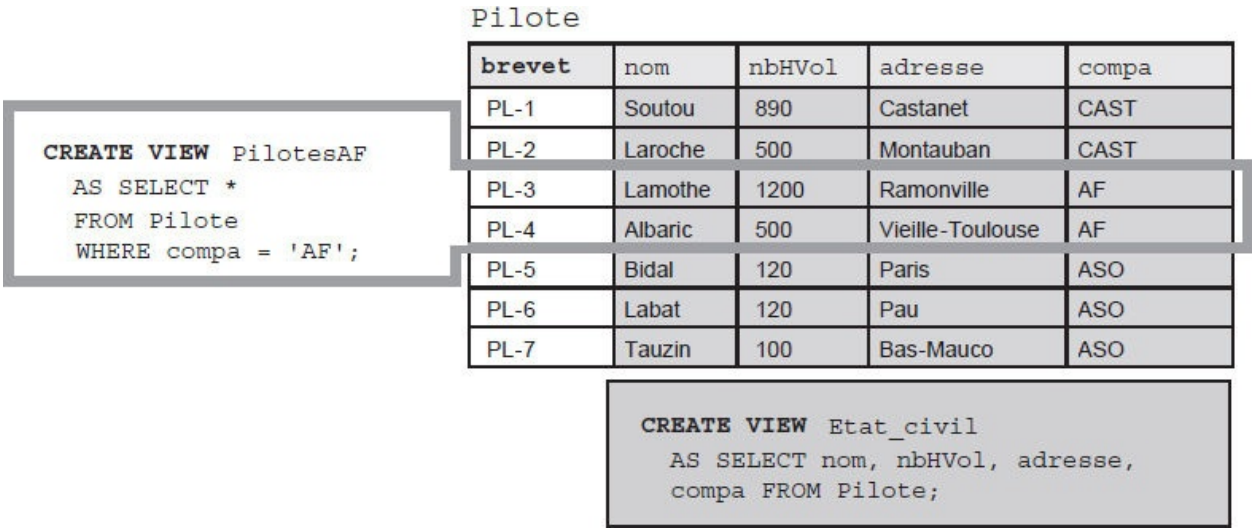


Une vue monotable est définie par une requête SELECT ne comportant qu'une seule table dans sa clause FROM.

Vues monotables

Les mécanismes présentés ci-après s'appliquent aussi, pour la plupart, aux vues multitables (étudiées plus loin). Considérons les deux vues illustrées par la figure suivante et dérivées de la table Pilote. La vue PilotesAF décrit les pilotes d'Air France à l'aide d'une restriction (éléments du WHERE). La vue Etat_civil est constituée par une projection de certaines colonnes (éléments du SELECT).

Figure 5-10 *Deux vues d'une table*



Une fois créée, une vue s'interroge comme une table par tout utilisateur, sous réserve qu'il ait obtenu le privilège en lecture directement (`GRANT SELECT ON` *nomVue* `TO`…). Le tableau suivant présente une interrogation des deux vues :

Tableau 5-14 Interrogation d'une vue

Besoin et requête	Résultat

Somme des heures de vol des pilotes d'Air France.
`SELECT SUM(nbHVol) FROM` **`PilotesAF`**`;`
	`+-------------+`
	`| SUM(nbHVol) |`
	`+-------------+`
	`| 1700.00 |`
	`+-------------+`

Nombre de pilotes.
`SELECT COUNT(*) FROM` **`Etat_civil`**`;`
	`+----------+`
	`| COUNT(*) |`
	`+----------+`
	`| 7 |`
	`+----------+`

À partir de cette table et de ces vues, nous allons étudier d'autres options de l'instruction `CREATE VIEW`.

Alias

Les alias, s'ils sont utilisés, désignent le nom de chaque colonne de la vue. Ce mécanisme permet de mieux contrôler les noms de colonnes. Quand un alias n'est pas présent, la colonne prend le nom de l'expression renvoyée par la requête de définition. Ce mécanisme sert à masquer les noms des colonnes de l'objet source.

Les vues suivantes sont créées avec des alias qui masquent le nom des colonnes de la table source. Les deux écritures sont équivalentes.

Tableau 5-15 Vue avec alias

Écriture 1	Écriture 2
`CREATE OR REPLACE VIEW PilotesPasAF (codepil,nomPil,heuresPil, adressePil, societe) AS SELECT * FROM Pilote WHERE NOT (compa = 'AF');`	`CREATE OR REPLACE VIEW PilotesPasAF AS SELECT brevet codepil, nom nomPil, nbHVol heuresPil, adresse adressePil, compa societe FROM Pilote WHERE NOT (compa = 'AF');`

Contenu de la vue :

```
+---------+---------+----------+------------+---------+
| codepil | nomPil  | heuresPil | adressePil | societe |
+---------+---------+----------+------------+---------+
| PL-1    | Soutou  |    890.00 | Castanet   | CAST    |
| PL-2    | Laroche |    500.00 | Montauban  | CAST    |
| PL-5    | Bidal   |    120.00 | Paris      | ASO     |
| PL-6    | Labat   |    120.00 | Pau        | ASO     |
| PL-7    | Tauzin  |    100.00 | Bas-Mauco  | ASO     |
+---------+---------+----------+------------+---------+
```

Vue d'une vue

L'objet source d'une vue est en général une table, mais peut aussi être une vue. La vue suivante est définie à partir de la vue `PilotesPasAF` précédemment créée. Notez qu'il aurait été possible d'utiliser des alias pour renommer les colonnes de la nouvelle vue.

Tableau 5-16 Vue d'une vue

Création	Contenu de la vue
`CREATE OR REPLACE VIEW EtatCivilPilotesPasAF AS SELECT nomPil,heuresPil,adressePil FROM PilotesPasAF;`	`+---------+----------+------------+ | nomPil | heuresPil | adressePil | +---------+----------+------------+ | Soutou | 890.00 | Castanet | | Laroche | 500.00 | Montauban | | Bidal | 120.00 | Paris | | Labat | 120.00 | Pau | | Tauzin | 100.00 | Bas-Mauco | +---------+----------+------------+`

Vues en lecture seule

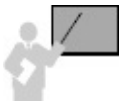
L'option `ALGORITHM=TEMPTABLE` déclare la vue non modifiable par `INSERT`, `UPDATE`, ou `DELETE`.

Redéfinissons une vue des pilotes n'étant pas d'Air France à l'aide de cette option. Les messages d'erreur induits par la clause de lecture seule, générés par MySQL, sont très parlants.

Tableau 5-17 Vue en lecture seule

Création	Opérations impossibles
`CREATE OR REPLACE ALGORITHM=TEMPTABLE VIEW PilotesPasAFRO AS SELECT * FROM Pilote WHERE NOT (compa = 'AF');`	`INSERT INTO PilotesPasAFRO VALUES ('PL-8','Ferry',5,'Paris','ASO');` **ERROR 1288** (HY000): The target table PilotesPasAFRO of the INSERT is not updatable `UPDATE PilotesPasAFRO SET nbHvol=nbHvol+2;` **ERROR 1288** (HY000): The target table PilotesPasAFRO of the UPDATE is not updatable `DELETE FROM PilotesPasAFRO;` **ERROR 1288** (HY000): The target table PilotesPasAFRO of the DELETE is not updatable

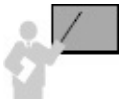
Vues modifiables



Lorsqu'il est possible d'exécuter des instructions INSERT, UPDATE ou DELETE sur une vue, cette dernière est dite « modifiable » (*updatable view*). Vous pouvez créer une vue qui est modifiable intrinsèquement.

Pour mettre à jour une vue, il doit exister une correspondance biunivoque entre les lignes de la vue et celles de l'objet source. De plus certaines conditions doivent être remplies.

Si une vue n'est pas modifiable, il n'est pas encore possible de programmer un déclencheur de type *instead of* qui prenne le pas sur l'instruction de modification de la vue, en spécifiant un bloc d'instructions à effectuer à la place. Les mises à jour de la vue seraient ainsi automatiquement répercutées au niveau d'une ou de plusieurs tables. Notons que ce type de déclencheur n'est arrivé qu'assez tardivement avec Oracle, DB2 et SQL Server.



Pour qu'une vue simple soit modifiable, sa requête de définition doit respecter les critères suivants :

- pas de directive DISTINCT, de fonction (AVG, COUNT, MAX, MIN, SUM, ou VARIANCE), d'expression dans le SELECT ;
- pas de GROUP BY, ORDER BY ou HAVING.

Dans notre exemple, nous constatons qu'il sera quand même possible d'ajouter un pilote à la vue Etat_civil, bien que la clé primaire de la table source ne soit pas renseignée. MySQL insère à la place, en l'absence de valeur par défaut de la clé primaire, la chaîne vide (' ') ; si la clé avait été une séquence, les insertions se feraient normalement. Cette opération ne pourra donc se faire qu'une seule fois, après, cela sera contradictoire avec la condition de correspondance biunivoque.

En revanche, il sera possible de modifier les colonnes de cette vue. On pourra aussi ajouter, modifier (sous réserve de respecter les éventuelles contraintes issues des colonnes de la table source), ou supprimer des pilotes en passant par la vue PilotesAF.

La dernière instruction est paradoxale, car elle permet d'ajouter un pilote de la compagnie 'ASO' en passant par la vue des pilotes de la compagnie 'AF'. La directive WITH CHECK OPTION permet d'éviter ces effets de bord indésirables pour l'intégrité de la base.

Tableau 5-18 Mises à jour de vues

Opérations possibles	Opérations non conseillées et impossibles

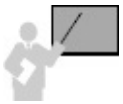
Suppression des pilotes de ASO
`DELETE FROM Etat_civil`
`WHERE compa = 'ASO';`
Le pilote *Lamothe* double ses heures
`UPDATE Etat_civil`
`SET nbHVol = nbHVol*2`
`WHERE nom = 'Lamothe';`
	Ajout d'un pilote (pas conseillé)
	`INSERT INTO Etat_civil VALUES`
	`('Raffarin',10,'Poitiers','ASO') ;`
	Ajout d'un autre pilote impossible
	`INSERT INTO Etat_civil VALUES`
	`('Lebur',20,'Bordeaux','ASO');`
	`ERROR 1062 (23000): Duplicate entry`
	`'' for key 1`

Ajout d'un pilote
`INSERT INTO PilotesAF VALUES`
`('PL-8','Ferry',5, 'Paris','AF');`
Modification
`UPDATE PilotesAF SET nbHVol = nbHVol*2;`
Suppression
`DELETE FROM PilotesAF WHERE nom='Ferry';`
Ajout d'un pilote qui n'est pas de
	Toute mise à jour qui ne respecterait pas les contraintes de la table Pilote.

'AF' !

```
INSERT INTO PilotesAF VALUES
 ('PL-9','Caboche', 600,'Rennes', 'ASO');
```

Directive `CHECK OPTION`



La directive `WITH CHECK OPTION` empêche un ajout ou une modification non conformes à la définition de la vue.

Interdisons l'ajout (ou la modification de la colonne `compa`) d'un pilote au travers de la vue `PilotesAF`, si le pilote n'appartient pas à la compagnie de code 'AF'. Il est nécessaire de redéfinir la vue `PilotesAF`. Le script suivant décrit la redéfinition de la vue, l'ajout d'un pilote et les tentatives d'ajout et de modification ne respectant pas les caractéristiques de la vue. Les messages sont très clairs.

Tableau 5-19 Vue avec `CHECK OPTION`

Opérations possibles	Opérations impossibles

Recréation de la vue
`CREATE OR REPLACE VIEW PilotesAF`
`  AS SELECT * FROM  pilote`
`  WHERE compa = 'AF'`
`WITH CHECK OPTION;`
	Ajout d'un pilote
	`INSERT INTO PilotesAF VALUES`
	`('PL-9','Caboche',600,'Rennes','ASO');`
	`ERROR 1369 (HY000): CHECK OPTION failed`
	`'bdsoutou.PilotesAF'`

Nouveau pilote
`INSERT INTO PilotesAF VALUES`
`('PL-11','Teste',900,'Revel', 'AF');`
`Query OK, 1 row affected (0.03 sec)`
	Modification de pilotes
	`UPDATE PilotesAF SET compa='ASO';`
	`ERROR 1369 (HY000): CHECK OPTION failed`
	`'bdsoutou.PilotesAF'`

Vues complexes

Une vue complexe est caractérisée par le fait qu'elle contient, dans sa définition, plusieurs tables (jointures) et une fonction appliquée à des regroupements ou à des expressions. La mise à jour de telles vues n'est pas toujours possible.

Pour pouvoir modifier une vue complexe, les restrictions sont les suivantes :

- La requête de définition ne doit pas contenir de sous-interrogation (jointure procédurale).
- Il n'est pas possible d'utiliser d'opérateur ensembliste (sauf `UNION [ALL]`).

La figure suivante présente deux vues complexes qui ne sont pas modifiables. La vue multitable `Pilotes_multi_AF` est créée à partir d'une jointure entre les tables `Compagnie` et `Pilote`. La vue `Moyenne_Heures_Pil` est créée à partir d'un regroupement de la table `Pilote`.

Figure 5-11 *Vues complexes*

Compagnie

comp	nrue	rue	ville	nomComp
AF	124	Port Royal	Paris	Air France
SING	7	Camparols	Singapour	Singapore AL

Pilote

brevet	nom	nbHVol	compa
PL-1	Ferrage	450	AF
PL-2	Guibert	900	AF
PL-3	Tisseyre	1000	SING

```
CREATE VIEW Pilotes_multi_AF
  AS SELECT p.brevet, p.nom,
            p.nbHVol, c.ville, c.nomComp
  FROM  Pilote p, Compagnie c
  WHERE p.compa = c.comp
  AND   p.compa = 'AF';
```

```
mysql> SELECT * FROM Pilotes_multi_AF;
+--------+---------+--------+-------+------------+
| brevet | nom     | nbHVol | ville | nomComp    |
+--------+---------+--------+-------+------------+
| PL-1   | Ferrage | 450.00 | Paris | Air France |
| PL-2   | Guibert | 900.00 | Paris | Air France |
+--------+---------+--------+-------+------------+
```

```
CREATE VIEW
       Moyenne_Heures_Pil
AS SELECT compa,
          AVG(nbHVol)
          moyenne
FROM      Pilote
GROUP BY compa;
```

```
mysql> SELECT *
    -> FROM  Moyenne_Heures_Pil;
+-------+-------------+
| compa | moyenne     |
+-------+-------------+
| AF    |  675.000000 |
| SING  | 1000.000000 |
+-------+-------------+
```

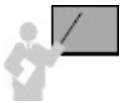
Mises à jour

Il semblerait qu'on ne puisse insérer aucun enregistrement dans ces vues du fait de la cohérence qu'il faudrait établir du sens *vue vers tables*. Les messages d'erreur générés par MySQL sont différents suivant la nature de la vue (monotable ou multitable). Nous verrons comment résoudre l'erreur du deuxième cas.

Tableau 5-20 Tentatives d'insertions dans des vues complexes

Vue monotable	Vue multitable
`INSERT INTO Moyenne_Heures_Pil VALUES ('TAT',50);` `ERROR 1288 (HY000): The target table Moyenne_Heures_Pil of the INSERT is not updatable`	`INSERT INTO Pilotes_multi_AF VALUES ('PL-4','Test',400,'Castanet','Castanet AL');` `ERROR 1394 (HY000): Can not insert into join view 'bdsoutou.Pilotes_multi_AF' without fields list`

On pourrait croire qu'il en est de même pour les modifications. Ce n'est pas le cas. Alors que la vue monotable `Moyenne_Heures_Pil` n'est pas modifiable, ni par `UPDATE` ni par `DELETE` (message d'erreur `1288`), la vue multitable `Pilotes_multi_AF` est transformable dans une certaine mesure, car la table `Pilote` (qui entre dans sa composition) est dite « protégée par clé » (*key preserved*). Nous verrons dans le prochain paragraphe la signification de cette notion.



Les colonnes de la vue correspondant à la table protégée par clé ne sont pas les seules à pouvoir être modifiées. Ici, `nbHVol` peut être mise à jour car elle appartient à la table protégée ; `ville` qui n'appartient pas à une table protégée peut aussi être modifée !

Les suppressions ne se répercutent pas sur les enregistrements de la table protégée par clé (`Pilote`).

Modifions et tentons de supprimer des enregistrements à travers la vue multitable `Pilotes_multi_AF`.

Tableau 5-21 Mise à jour d'une vue multitable

Mise à jour	Résultats

`-- Multiplie par 2 le nombre d'heures`
`UPDATE Pilotes_multi_AF`
	`SELECT brevet,nom,nbHVol`
	`FROM Pilotes_multi_AF;`
	`+--------+------------+---------+`
	`| brevet | nom | nbHVol |`

```
      SET nbHVol =  nbHVol * 2;
Query OK, 2 rows affected
Rows matched: 2 Changed: 2 Warnings: 0
```

```
+--------+------------+---------+
| PL-1   | Ferrage    |  900.00 |
| PL-2   | Guibert    | 1800.00 |
+--------+------------+---------+
```

```
-- Modif de la ville de la compagnie
UPDATE Pilotes_multi_AF
       SET ville =  'Orly';
```

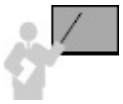
```
SELECT brevet,nom,ville
       FROM Pilotes_multi_AF;
+--------+------------+-------+
| brevet | nom        | ville |
+--------+------------+-------+
| PL-1   | Ferrage    | Orly  |
| PL-2   | Guibert    | Orly  |
+--------+------------+-------+
```

```
SELECT comp,ville,nomComp
       FROM Compagnie;
+------+----------+--------------+
| comp | ville    | nomComp      |
+------+----------+--------------+
| AF   | Orly     | Air France   |
| SING | Singapour | Singapore AL |
+------+----------+--------------+
```

```
--Pas possible :
DELETE FROM Pilotes_multi_AF;
```

```
ERROR 1395 (HY000): Can not delete from
join view 'bdsoutou.Pilotes_multi_AF'
```

Tables protégées (key preserved tables)



Une table est dite protégée par sa clé (*key preserved*) si sa clé primaire est préservée dans la clause de jointure et se retrouve en tant que colonne de la vue multitable (elle peut jouer le rôle de clé primaire de la vue).

En considérant les données initiales pour la vue multitable `Vue_Multi_Comp_Pil`, la table préservée est la table `Pilote`, car la colonne `brevet` identifie chaque enregistrement extrait de la vue, alors que la colonne `comp` ne le fait pas.

Tableau 5-22 Vue multitable

Création de la vue	Résultats

```
CREATE VIEW
Vue_Multi_Comp_Pil
 AS SELECT c.comp,
     c.nomComp,
      p.brevet,
p.nom,
     p.nbHVol
 FROM  Pilote p,
       Compagnie c
```

```
+------+--------------+--------+------------+---------+
| comp | nomComp      | brevet | nom        | nbHVol  |
+------+--------------+--------+------------+---------+
| AF   | Air France   | PL-1   | Ferrage    |  450.00 |
| AF   | Air France   | PL-2   | Guibert    |  900.00 |
| SING | Singapore AL | PL-3   | Tisseyre   | 1000.00 |
+------+--------------+--------+------------+---------+
```

```
 WHERE
p.compa=c.comp;
```

Cela ne veut pas dire pour autant que cette vue est modifiable. Étudions à présent les conditions qui régissent ces limitations.

Critères

Une vue multitable modifiable (*updatable join view* ou *modifiable join view*) est une vue qui n'est pas définie avec l'option `ALGORITHM=TEMPTABLE` et qui est telle que la requête de définition contient plusieurs tables dans la clause `FROM`.

Aucune suppression n'est possible.

Les insertions sont permises seulement en isolant toutes les colonnes d'une seule table source.

Attention aux effets de bord quand vous modifiez une colonne provenant d'une table non protégée par clé. Il est plus naturel de modifier directement la table en question.

Modifions de différentes manières la vue multitable `Vue_Multi_Comp_Pil`. La première tente une suppression, les deux suivantes modifient tantôt une colonne de la table protégée, tantôt une colonne de la table non protégée. Les deux dernières instructions insèrent dans chacune des deux tables.

Tableau 5-23 Mises à jour

Mise à jour	Résultats
`DELETE FROM Vue_Multi_Comp_Pil` `WHERE comp='AF';`	`ERROR 1395 (HY000): Can not delete from join view 'bdsoutou.Vue_Multi_Comp_Pil'`

`UPDATE Vue_Multi_Comp_Pil`
`SET nbHVol = nbHVol * 3;`
`Query OK, 3 rows affected (0.10 sec)`
`Rows matched: 3 Changed: 3 Warnings: 0`
	`SELECT brevet, nbHVol FROM Pilote;`
	`+--------+---------+`
	`| brevet | nbHVol |`
	`+--------+---------+`
	`| PL-1 | 1350.00 |`
	`| PL-2 | 2700.00 |`
	`| PL-3 | 3000.00 |`
	`+--------+---------+`

	`(SELECT comp,ville,nomComp` `FROM Compagnie;`

```
UPDATE Vue_Multi_Comp_Pil
       SET nomComp = 'Dupond';
Query OK, 2 rows affected (0.38 sec)
Rows matched: 3  Changed: 2  Warnings: 0
```

```
+------+-----------+---------+
| comp | ville     | nomComp |
+------+-----------+---------+
| AF   | Paris     | Dupond  |
| SING | Singapour | Dupond  |
+------+-----------+---------+
```

```
INSERT INTO Vue_Multi_Comp_Pil
  (brevet,nom,nbHVol)
  VALUES ('PL-5', 'Jean',2500);;
```

```
SELECT brevet,nom,nbHVol
       FROM Pilote;
+--------+------------+---------+
| brevet | nom        | nbHVol  |
+--------+------------+---------+
| PL-1   | Ferrage    | 1350.00 |
| PL-2   | Guibert    | 2700.00 |
| PL-3   | Tisseyre   | 3000.00 |
| PL-5   | Jean       | 2500.00 |
+--------+------------+---------+
```

```
INSERT INTO Vue_Multi_Comp_Pil
   (comp,nomComp)
    VALUES ('TAT', 'Test');
```

```
SELECT comp,ville,nomComp
       FROM Compagnie;
+------+-----------+---------+
| comp | ville     | nomComp |
+------+-----------+---------+
| AF   | Paris     | Dupond  |
| SING | Singapour | Dupond  |
| TAT  | Paris     | Test    |
+------+-----------+---------+
```

Autres utilisations de vues

Les vues peuvent également servir pour simplifier l'écriture de requêtes complexes, renforcer la confidentialité et une certaine intégrité.

Simplifier les noms

Une vue permet de simplifier la manipulation d'une table ayant un nom long ou des colonnes portant des noms compliqués. Considérons par exemple la table COLLATION_CHARACTER_SET_APPLICABILITY[COLLATION_NAME,CHARACTER_SET_NAME] qui décrit les collations des jeux de caractères disponibles, et qui est située dans la base de données stockant le dictionnaire des données (INFORMATION_SCHEMA). Nous étudierons dans la prochaine section les différentes tables de cette base système. Supposons que l'on désire souvent accéder à cette table pour connaître les différentes collations possibles pour les jeux de caractères latins.

Créons la vue CollationsLatines qui simplifie l'accès à cette table au niveau du nom, mais aussi au niveau des colonnes. Interrogeons cette vue de manière à connaître les collations spécifiques à la langue de Molière ou à celle de Goethe.

Tableau 5-24 Vue pour simplifier les noms

Création	Interrogation

`CREATE VIEW CollationsLatines (collation,jeu) AS SELECT * FROM INFORMATION_SCHEMA.↵ COLLATION_CHARACTER_SET_APPLICABILITY WHERE CHARACTER_SET_NAME LIKE 'Latin%';`	`SELECT * FROM CollationsLatines WHERE collation LIKE '%french%' OR collation LIKE '%german%';`
	`+-------------------+--------+`
	`| collation | jeu |`
	`+-------------------+--------+`
	`| latin1_german1_ci | latin1 |`
	`| latin1_german2_ci | latin1 |`
	`+-------------------+--------+`

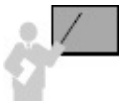
On dira que MySQL est plus « branché » par la nouveauté, Goethe étant né 76 ans après le décès de Molière. Aucun `french` dans la base, donc.

Contraintes de vérification

Nous avons décrit au chapitre 1 les contraintes de vérification (CHECK) qui ne sont pas encore totalement prises en charge. Il est possible de programmer ce type de contraintes par des vues.

Considérons la table `Pilote` illustrée ci-après et programmons, par l'intermédiaire de la vue `VueGradePilotes`, la contrainte vérifiant qu'un pilote :

- ne peut être commandant de bord qu'à la condition qu'il ait entre 1 000 et 4 000 heures de vol ;
- ne peut être copilote qu'à la condition qu'il ait entre 100 et 1 000 heures de vol ;
- ne peut être instructeur qu'à partir de 3 000 heures de vol.



Les règles conseillées pour manipuler les enregistrements d'une vue *v1* décrivant des contraintes de vérification sur une table *t1* sont les suivantes :

- modification et insertion par la vue *v1* ;
- suppression et lecture par la table *t1*.

Figure 5-12 *Vue simulant la contrainte* CHECK

Pilote

brevet	nom	nbHVol	grade
PL-1	Vielle	1000	CDB
PL-2	Treilhou	450	COPI
PL-3	Filoux	9000	INST
PL-4	Minier	1000	COPI

```
CREATE VIEW VueGradePilotes
 AS SELECT brevet,nom,nbHVol,grade
 FROM Pilote
  WHERE (grade ='CDB'
        AND nbHVol BETWEEN 1000 AND 4000)
  OR    (grade = 'COPI'
         AND nbHVol BETWEEN 100 AND 1000)
  OR    (grade = 'INST' AND nbHVol > 3000)
  WITH CHECK OPTION;
```

Manipulons à présent la vue de notre exemple :

Tableau 5-25 Manipulations des vues pour l'intégrité référentielle

Mises à jour possibles	**Mises à jour non valides :** `ERROR 1369 (HY000): CHECK OPTION failed 'bdsoutou.VueGradePilotes'`

`INSERT INTO VueGradePilotes (brevet,nom,nbHVol,grade)VALUES ('PL-1', 'Vielle', 1000, 'CDB');`
`INSERT INTO VueGradePilotes (brevet,nom,nbHVol,grade) VALUES ('PL-2', 'Treihlou', 450, 'COPI');`
`INSERT INTO VueGradePilotes (brevet,nom,nbHVol,grade) VALUES ('PL-3', 'Filoux', 9000, 'INST');`
`INSERT INTO VueGradePilotes (brevet,nom,nbHVol,grade) VALUES ('PL-4', 'Minier', 1000, 'COPI');`
	`INSERT INTO VueGradePilotes (brevet,nom,nbHVol,grade)VALUES ('PL-5','Trop jeune',100,'CDB');`
	`INSERT INTO VueGradePilotes (brevet,nom,nbHVol,grade) VALUES ('PL-6', 'Inexperimente',2999,'INST');`

`UPDATE VueGradePilotes SET grade ='CDB' WHERE brevet = 'PL-4';`	`UPDATE VueGradePilotes SET grade ='INST' WHERE brevet = 'PL-2';`
	`UPDATE VueGradePilotes SET nbHVol= 50 WHERE brevet = 'PL-3';`

Confidentialité

La confidentialité est une des vocations premières des vues. Outre l'utilisation de variables d'environnement, il est possible de restreindre l'accès à des tables en fonction de moments précis.

Les vues suivantes limitent temporellement les accès en lecture et en écriture à des tables.

Notez qu'il est possible, en plus, de limiter l'accès à un utilisateur particulier en utilisant une variable d'environnement (exemple : rajout de la condition `AND CURRENT_USER() = 'Paul@localhost'` à une vue).

Tableau 5-26 Vues pour restreindre l'accès à des moments précis

Définition de la vue	Accès

`CREATE VIEW VueDesCompagniesJoursFeries`
`  AS SELECT * FROM Compagnie`
`  WHERE DATE_FORMAT(SYSDATE(),'%W')`
`  IN ('Sunday','Saturday');`
	Restriction en lecture de la table `Compagnie`, les samedis et dimanches. Mises à jour possibles à tout moment.

`CREATE VIEW VueDesPilotesJoursOuvrables`
`  AS SELECT * FROM Pilote`
`  WHERE CURTIME()+0`
`  BETWEEN 83000 AND 173000`
`  AND DATE_FORMAT(SYSDATE(),'%W')`
`  NOT IN ('Sunday', 'Saturday')`
`  WITH CHECK OPTION;`
	Restriction, en lecture et en écriture (à cause de `WITH CHECK OPTION`), de la table `Pilote` les jours ouvrables de 8h30 à 17h30.

Transmission de droits

Les mécanismes de transmission et de révocation de privilèges que nous avons étudiés s'appliquent également aux vues. Ainsi, si un utilisateur désire transmettre des droits sur une partie d'une de ses tables, il utilisera une vue. Seules les données appartenant à la vue seront accessibles aux bénéficiaires.

Les privilèges objet qu'il est possible d'attribuer sur une vue sont les mêmes que ceux applicables sur les tables (`SELECT`, `INSERT`, `UPDATE` sur une ou plusieurs colonnes, `DELETE`).

Tableau 5-27 Privilèges sur les vues

Attribution du privilège	Signification
`GRANT SELECT ON VueDesCompagniesJoursFeries` `  TO 'Paul'@'localhost';`	Accès en local de l'utilisateur `Paul` en lecture sur la vue

	VueDesCompagniesJoursFériés.
`GRANT INSERT ON VueDesPilotesJoursOuvrables TO 'Jules'@'localhost';`	Accès en local de l'utilisateur Jules en écriture sur la vue VueDesCompagniesJoursFériés.

Modification d'une vue [ALTER VIEW]

Vous devez au moins posséder les privilèges CREATE VIEW et DELETE au niveau d'une vue pour pouvoir la modifier. La syntaxe SQL est la suivante :

```
ALTER [ALGORITHM = {UNDEFINED | MERGE | TEMPTABLE}]
    VIEW [nomBase.]nomVue [(listecolonnes)]
    AS requêteSELECT
    [WITH [CASCADED | LOCAL] CHECK OPTION];
```

Les transformations peuvent concerner toutes les parties d'une vue existante : la politique de création (ALGORITHM), la liste des colonnes, la requête, etc. Voir la section « Création d'une vue ».

Visualisation d'une vue [SHOW CREATE VIEW]

Pour pouvoir visualiser la requête de définition d'une vue, l'instruction que MySQL propose est la suivante :

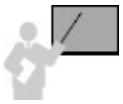
```
SHOW CREATE VIEW  [nomBase.]nomVue;
```

En arrangeant l'état de sortie, vous pouvez découvrir comment MySQL stocke la définition de la vue précédemment créée :

```
SHOW CREATE VIEW VueDesCompagniesJoursFeries;
+----------------------------+---------------------------------------------+
| View                       | Create View                                 |
+----------------------------+---------------------------------------------+
| VueDesCompagniesJoursFeries | CREATE ALGORITHM=UNDEFINED                  |
|                            | DEFINER=`root`@`localhost` SQL SECURITY     |
|                            | DEFINER VIEW `VueDesCompagniesJoursFeries` AS |
|                            | select sql_no_cache `Compagnie`.`comp` AS   |
|                            | `comp`,`Compagnie`.`nrue` AS                |
|                            | `nrue`,`Compagnie`.`rue` AS                 |
|                            | `rue`,`Compagnie`.`ville` AS                |
|                            | `ville`,`Compagnie`.`nomComp` AS `nomComp`  |
|                            | from `Compagnie` where                      |
|                            | (date_format(sysdate(),_latin1'%W') in      |
|                            | (_latin1'Sunday',_latin1'Saturday'))        |
+----------------------------+---------------------------------------------+
```

Suppression d'une vue [DROP VIEW]

Vous devez posséder le privilège DROP sur une vue pour pouvoir la supprimer.



La suppression d'une vue n'entraîne pas la destruction des données qui résident toujours dans les tables.

La syntaxe SQL est la suivante :

```
DROP VIEW [IF EXISTS]
          [nomBase.]nomVue [,nomBase2.]nomVue2...
          [RESTRICT | CASCADE];
```

- IF EXISTS évite une erreur dans le cas où la vue n'existe pas.
- RESTRICT et CASCADE ne sont pas encore opérationnels, il concerneront probablement la répercussion de la suppression entre vues interdépendantes.

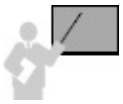
Dictionnaire des données

Le dictionnaire des données (*metadata* ou *data dictionary*) est une partie majeure d'une base de données MySQL qu'on peut assimiler à une structure centralisée.

Selon la version de MySQL, le dictionnaire des données est composé d'une vingtaine à une quarantaine de vues (issues de tables système non visibles). Les versions antérieures à 5.0 (3.23, 4.0 et 4.1) ne mentionnaient même pas ces vues dans la documentation. Depuis la version 5, chaque nouvelle mouture du serveur apporte son lot de nouveautés notamment concernant le moteur de stockage InnoDB (18 vues en version 5.0, 40 en version 5.5, une soixantaine en version 5.7).

Ces vues, qui sont appelées *tables* par abus de langage dans la documentation officielle, dans les livres et sur les forums[1] sont situées dans la base INFORMATION_SCHEMA. Elles permettent de stocker toute information décrivant tous les objets contenus dans toute base de données. Les noms des vues se rapprochent de la spécification ANSI/ISO SQL:2003 standard Part 11

Schemata. MySQL se rapproche davantage de la norme SQL (et des SGBD IBM DB2 et Microsoft SQL Server), plutôt que d'Oracle dont le dictionnaire de données est propriétaire (composé de plus de 600 vues).

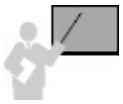


Avant la version 8, le dictionnaire des données était stocké dans des fichiers non transactionnels. Depuis, les tables système sont implémentées avec un moteur InnoDB et les vues restent toujours accessibles par requêtes.

Constitution

Le dictionnaire des données contient :

- la définition des tables, vues, index, séquences, procédures, fonctions et déclencheurs ;
- la description de l'espace disque alloué et occupé par chaque objet ;
- les valeurs par défaut des colonnes (`DEFAULT`) ;
- la description des contraintes d'intégrité référentielle (de vérification à venir) ;
- le nom des utilisateurs de la base ;
- les privilèges pour chaque utilisateur ;
- des informations d'audit (accès aux objets) et d'autre nature (commentaires, par exemple).



Toutes les tables du dictionnaire des données ne sont accessibles qu'en lecture seulement. Elles appartiennent à la base de données `INFORMATION_SCHEMA`. L'interrogation du dictionnaire des données est permise à tout utilisateur (qui ne verra que les objets qui lui sont toutefois accessibles avec ses propres privilèges) et peut se faire au travers de requêtes `SELECT` ou par le biais de la commande `SHOW`.

Toutes les informations contenues dans les tables du dictionnaire des données

sont codées en `minuscules`.

Le dictionnaire des données est mis automatiquement à jour après chaque instruction SQL du LMD (`INSERT`, `UPDATE`, `DELETE`, `LOCK TABLE`).

Les avantages d'interroger le dictionnaire des données par des requêtes sont les suivants :

- Conforme aux règles d'E.F. Codd (le père du modèle relationnel) ; toutes les manipulations sont réalisées à l'aide des opérateurs relationnels sur des tables.
- Inutile d'apprendre de nouvelles instructions (`SHOW` *`paramètres`*) propriétaires de MySQL. La migration vers un autre SGBD est ainsi facilitée.
- Les possibilités d'extraction sont quasiment illimitées du fait du grand nombre de tables et de jointures (ou d'autres opérateurs) possible.
- Vous avez tellement souffert au chapitre 4, que vous avez ici l'occasion de mettre à l'épreuve vos connaissances dans un contexte plus « système ».

Modèle graphique du dictionnaire des données

La figure 5-11 décrit le modèle relationnel (structure des vues) du dictionnaire des données de la version MySQL 5.0 (issu du site http://www.xcdsql.org/Misc/MySQL_INFORMATION_SCHEMA.html). Vous trouverez aussi une représentation (cliquable) des vues du dictionnaire de la version 5.1 à l'adresse suivante : http://www.xcdsql.org/MySQL/information_schema/5.1/MySQL_5_1_INFORM

Démarche à suivre

La démarche à suivre afin d'interroger correctement le dictionnaire des données à propos d'un objet est la suivante :

- trouver le nom de la vue (ou des vues) pertinente(s) à partir du schéma précédent ;
- choisir les colonnes de la vue (ou des vues) à sélectionner (soit à partir du graphique, soit en affichant la structure de la vue par la commande `DESCRIBE`) ;

- interroger la vue (ou les vues) en exécutant une instruction `SELECT` contenant les colonnes intéressantes.

Recherche du nom d'une vue

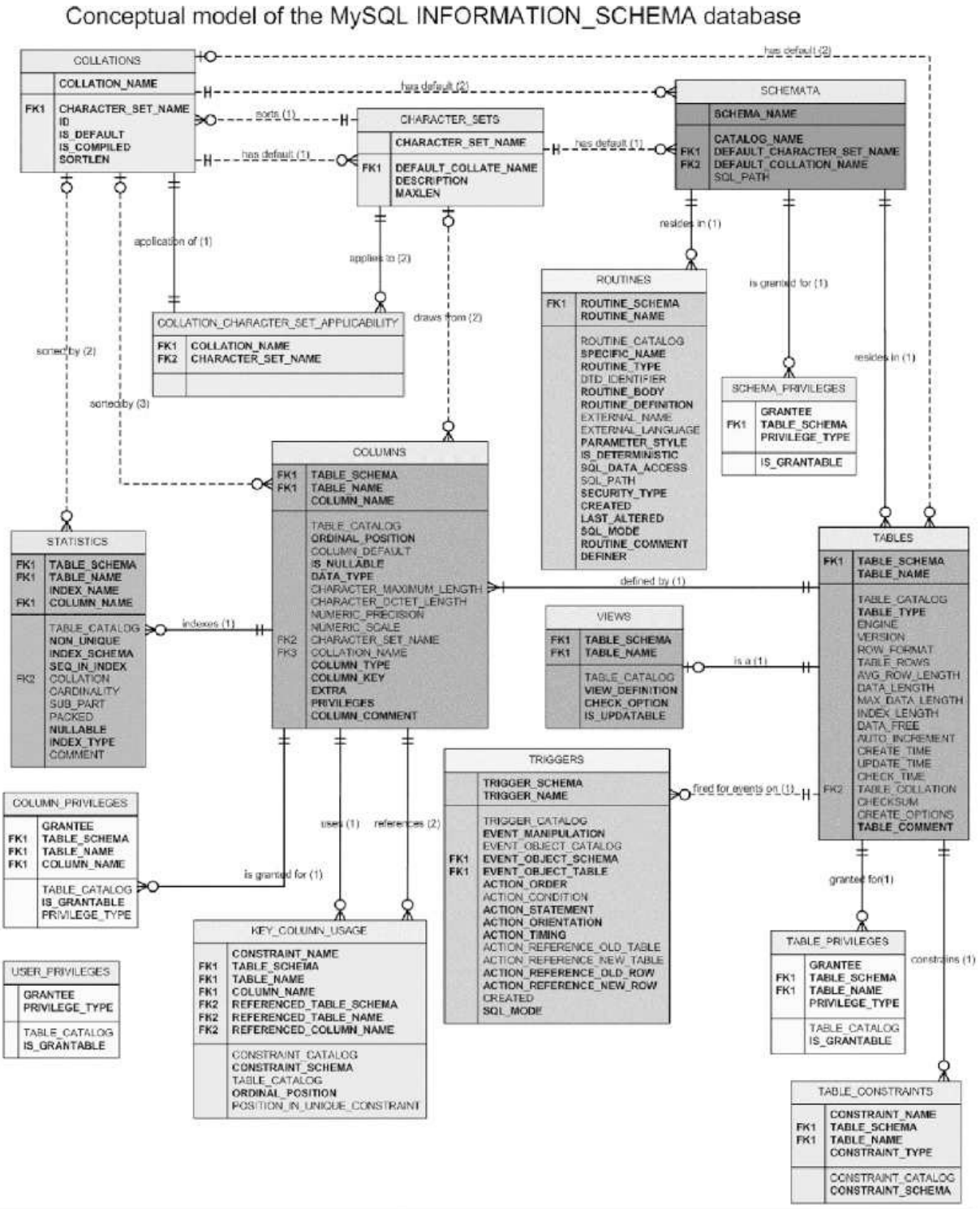
Il n'existe pas de moyen automatique de trouver le nom d'une vue pertinente en fonction de l'information que l'on cherche (relative au serveur, aux utilisateurs, aux index, etc.). Cela dit, pour une classification donnée (serveur, utilisateur, table, contrainte, etc.), vous trouverez le plus souvent moins de cinq vues associées.

Choisir les colonnes

Le choix des colonnes d'une vue du dictionnaire des données s'effectue après avoir listé sa structure (par `DESCRIBE`). Le nom de chaque colonne est en général assez parlant.

Supposons que notre recherche concerne les vues des tables existantes dans différents schémas. La vue système à interroger est `INFORMATION_SCHEMA.VIEWS` qui contient huit colonnes. Il apparaît que la clause de définition de chaque vue est contenue dans la colonne `VIEW_DEFINITION`. La colonne `CHECK_OPTION` doit en principe indiquer que la vue est déclarée avec une contrainte de vérification. Mais quelle colonne renseigne donc le nom de chaque vue ? Nous verrons qu'il s'agit de `TABLE_NAME`.

Figure 5-13 *Modèle graphique du dictionnaire des données*




```
DESCRIBE INFORMATION_SCHEMA.VIEWS;
+-----------------+--------------+------+-----+---------+-------+
| Field           | Type         | Null | Key | Default | Extra |
```

```
+-----------------+--------------+------+-----+---------+-------+
| TABLE_CATALOG   | varchar(512) | NO   |     |         |       |
| TABLE_SCHEMA    | varchar(64)  | NO   |     |         |       |
| TABLE_NAME      | varchar(64)  | NO   |     |         |       |
| VIEW_DEFINITION | longtext     | NO   |     | NULL    |       |
| CHECK_OPTION    | varchar(8)   | NO   |     |         |       |
| IS_UPDATABLE    | varchar(3)   | NO   |     |         |       |
| DEFINER         | varchar(77)  | NO   |     |         |       |
| SECURITY_TYPE   | varchar(7)   | NO   |     |         |       |
+-----------------+--------------+------+-----+---------+-------+
```

Interroger la vue

L'interrogation de la vue sur les colonnes choisies est l'étape finale de la recherche de données dans le dictionnaire. Il convient d'écrire une requête monotable ou multitable (jointures) qui extrait des données contenues dans la vue. Ces données sont en fait contenues dans des tables système qui ne sont pas accessibles pour des raisons sécuritaires.

Supposons que je sois `root` en local et que je désire connaître le nom, l'emplacement et le caractère contraint de toutes les vues existantes :

```
SELECT TABLE_SCHEMA,TABLE_NAME,CHECK_OPTION FROM INFORMATION_SCHEMA.VIEWS;
+--------------+-----------------------------+--------------+
| TABLE_SCHEMA | TABLE_NAME                  | CHECK_OPTION |
+--------------+-----------------------------+--------------+
| bdnouvelle   | VueDesSocietes              | NONE         |
| bdsoutou     | VueDesPilotesJoursOuvrables | CASCADED     |
+--------------+-----------------------------+--------------+
```

Si j'avais voulu connaître les vues contenues seulement dans la base `bdsoutou`, il suffisait d'ajouter la condition (`WHERE TABLE_SCHEMA='bdsoutou'`).

Vous pouvez noter que MySQL utilise :

- La colonne `TABLE_SCHEMA` pour désigner une *database*.
- La colonne `TABLE_NAME` pour stocker le nom de chaque vue des différents schémas. Ici la norme SQL doit y être pour quelque chose (Oracle nomme la colonne `VIEW_NAME`).
- La colonne `CHECK_OPTION` pour indiquer le caractère restreint de chaque vue (la première n'est pas restreinte, la seconde l'est).

Classification des vues

Le tableau suivant classifie les vues selon leur fonctionnalité. Notez qu'aucune redondance ni de synonyme n'existent (si vous voulez réaliser une extraction

pour découvrir quelque chose, il n'y aura pas beaucoup de requêtes différentes possibles).

Tableau 5-28 Vues du dictionnaire des données (version 5.5)

Nature de l'objet	Vues
Serveur	SCHEMATA : caractéristiques du serveur (jeux de caractères utilisés). ENGINE : moteurs du serveur. PLUGINS : composants logiciels et additifs au serveur. EVENTS : événements planifiés. GLOBAL_STATUS et SESSION_STATUS : variables serveur globales et de session. PROCESSLIST : sessions en cours. CHARACTER_SETS : informations sur les colonnes pour lesquelles l'utilisateur a reçu une autorisation. COLLATIONS et COLLATION_CHARACTER_SET_APPLICABILITY : relatifs aux jeux de caractères.
Privilèges	SCHEMA_PRIVILEGES : liste des prérogatives au niveau *database*. TABLE_PRIVILEGES : liste des prérogatives au niveau *table*. USER_PRIVILEGES : liste des prérogatives au niveau *user*. COLUMN_PRIVILEGES : liste des prérogatives au niveau *columns*.
Tables, séquences et partitions	TABLES : caractéristiques des tables (et séquences) dans les bases. PARTITIONS : caractéristiques des tables partitionnées.
Colonnes	COLUMNS : colonnes des tables et vues.
Index	STATISTICS : description des index.
Contraintes	TABLE_CONSTRAINTS : définition des contraintes de tables. KEY_COLUMN_USAGE : composition des contraintes (colonnes). REFERENTIAL_CONSTRAINTS : détail des contraintes référentielles (clés étrangères).
Vues	VIEWS : description des vues.
Sous-programmes	ROUTINES : description des sous-programmes stockés. PARAMETERS : paramètres des sous-programmes stockés.

TRIGGERS : description des déclencheurs.

Le moteur InnoDB (en principe celui mis en œuvre par défaut dans vos versions téléchargées) induit des vues additionnelles qui servent deux objectifs :

- monitoring de l'activité, détection de problèmes de performance (basé sur le taux de compression des tables, transactions et verrous). Les vues relatives à cet objectif sont : INNODB_CMP, INNODB_CMP_RESET, INNODB_CMPMEM, INNODB_CMPMEM_RESET, INNODB_TRX, INNODB_LOCKS et INNODB_LOCK_WAITS.
- extraction d'information concernant les objets manipulés par le moteur InnoDB. Les vues relatives à cet objectif sont : INNODB_SYS_TABLES, INNODB_SYS_INDEXES, INNODB_SYS_COLUMNS, INNODB_SYS_FIELDS, INNODB_SYS_FOREIGN, INNODB_SYS_FOREIGN_COLS et INNODB_SYS_TABLESTATS.

Interrogeons à présent quelques-unes de ces vues dans le cadre d'exemples concrets.

Moteurs du serveur

La requête suivante interroge la vue ENGINES et permet de connaître les moteurs disponibles sur votre serveur MySQL.

```
SELECT ENGINE, SUPPORT, COMMENT
  FROM INFORMATION_SCHEMA.ENGINES;
+--------------------+---------+----------------------------------------------------+
| ENGINE             | SUPPORT | COMMENT                                            |
+--------------------+---------+----------------------------------------------------+
| FEDERATED          | NO      | Federated MySQL storage engine                     |
| MRG_MYISAM         | YES     | Collection of identical MyISAM tables              |
| MyISAM             | YES     | Default engine as of MySQL 3.23 great performance |
| BLACKHOLE          | YES     | /dev/null storage engine …                         |
| CSV                | YES     | CSV storage engine                                 |
| MEMORY             | YES     | Hash based, stored in memory, …                    |
| ARCHIVE            | YES     | Archive storage engine                             |
| InnoDB             | DEFAULT | Supports transactions, row-level locking, …        |
| PERFORMANCE_SCHEMA | YES     | Performance Schema                                 |
+--------------------+---------+----------------------------------------------------+
```

Les colonnes de cette vue sont les suivantes :

- ENGINE désigne le nom du moteur.
- SUPPORT précise si le moteur en question est disponible sur le serveur.
- COMMENT décrit brièvement le moteur.

- XA (YES, NO ou NULL) indique la compatibilité avec la spécification XA (transactions distribuées).
- SAVEPOINTS (valant YES, NO ou NULL) indique si les points de validation sont permis dans la programmation.

Bases de données du serveur

La requête suivante interroge la vue SCHEMATA et permet de retrouver les caractéristiques (jeux de caractères) des bases de données hébergées par le serveur.

```
SELECT SCHEMA_NAME AS 'Base de donnees',
       DEFAULT_CHARACTER_SET_NAME AS 'Jeu caracteres',
       DEFAULT_COLLATION_NAME AS 'Collation'
       FROM INFORMATION_SCHEMA.SCHEMATA;
+--------------------+----------------+--------------------+
| Base de donnees    | Jeu caracteres | Collation          |
+--------------------+----------------+--------------------+
| information_schema | utf8           | utf8_general_ci    |
| bdnouvelle         | ascii          | ascii_general_ci   |
| bdsoutou           | latin1         | latin1_swedish_ci  |
| mysql              | latin1         | latin1_swedish_ci  |
| test               | latin1         | latin1_swedish_ci  |
+--------------------+----------------+--------------------+
```

Notez que MySQL utilise :

- la colonne DEFAULT_CHARACTER_SET_NAME pour désigner le jeu de caractères d'une *database*. Le dictionnaire est lui-même codé en utf8.
- la colonne DEFAULT_COLLATION_NAME pour désigner la collation du jeu de caractères.

Les colonnes CATALOG_NAME et SQL_PATH, qui proviennent de la norme SQL, ne sont pas renseignées : la première de ces colonnes est relative au concept de schéma, et la seconde à une routine (sous-programme).

Composition d'une base

La requête suivante interroge la vue TABLES et décrit la composition des bases de

données utilisateur (j'ai filtré volontairement les lignes qui correspondent aux bases de MySQL).

```
SELECT TABLE_SCHEMA,TABLE_NAME, TABLE_TYPE,DATE(CREATE_TIME)
       FROM INFORMATION_SCHEMA.TABLES
       WHERE TABLE_SCHEMA NOT IN ('information_schema','test','mysql');
+--------------+----------------------------+------------+------------------+
| TABLE_SCHEMA | TABLE_NAME                 | TABLE_TYPE | DATE(CREATE_TIME) |
+--------------+----------------------------+------------+------------------+
| bdnouvelle   | VueDesSocietes             | VIEW       | NULL              |
| bdsoutou     | Installer                  | BASE TABLE | 2005-11-30        |
| bdsoutou     | Logiciel                   | BASE TABLE | 2005-11-30        |
| bdsoutou     | PCSeuls                    | BASE TABLE | 2005-11-30        |
| bdsoutou     | Pilote                     | BASE TABLE | 2005-11-30        |
| bdsoutou     | Poste                      | BASE TABLE | 2005-11-30        |
| bdsoutou     | Salle                      | BASE TABLE | 2005-11-30        |
| bdsoutou     | Segment                    | BASE TABLE | 2005-11-30        |
| bdsoutou     | Softs                      | BASE TABLE | 2005-11-30        |
| bdsoutou     | Types                      | BASE TABLE | 2005-11-30        |
| bdsoutou     | VueDesPilotesJoursOuvrables | VIEW       | NULL              |
+--------------+----------------------------+------------+------------------+
```

Vous pouvez remarquer que MySQL utilise :

- la colonne `TABLE_TYPE` pour désigner le type de la structure de stockage (les tables temporaires, si elles existent, n'apparaissent pas).
- la colonne `CREATE_TIME` pour désigner la date de création de l'objet.

Détail de stockage d'une base

En utilisant la même vue du dictionnaire, intéressons-nous à la table `Installer` dans la base `bdsoutou` qui fait partie du schéma des exercices de ce livre. La requête suivante extrait des informations intéressantes.

```
SELECT ENGINE,AUTO_INCREMENT,TABLE_ROWS,AVG_ROW_LENGTH,DATA_LENGTH
       FROM INFORMATION_SCHEMA.TABLES
       WHERE TABLE_SCHEMA = 'bdsoutou' AND TABLE_NAME='Installer';
+--------+----------------+------------+----------------+-------------+
| ENGINE | AUTO_INCREMENT | TABLE_ROWS | AVG_ROW_LENGTH | DATA_LENGTH |
+--------+----------------+------------+----------------+-------------+
| InnoDB |             15 |         14 |           1170 |       16384 |
+--------+----------------+------------+----------------+-------------+
```

Notez que MySQL utilise :

- la colonne `ENGINE` pour désigner le type de moteur de stockage de la table en question.
- la colonne `AUTO_INCREMENT` pour désigner la prochaine valeur de la séquence.
- la colonne `TABLE_ROWS` pour donner le nombre d'enregistrements de la table

(ici c'est bien cohérent avec la séquence qui fait office de clé primaire).

- la colonne AVG_ROW_LENGTH pour désigner la taille moyenne d'une ligne en octets.
- la colonne DATA_LENGTH pour désigner la taille de la table en octets.

Citons, pour en terminer avec cette vue, les colonnes :

- TABLE_COLLATION qui indique le jeu de caractères de la table.
- TABLE_COMMENT qui renseigne notamment à propos des références entre tables par les clés étrangères.

Structure d'une table

Intéressons-nous à présent à la vue COLUMNS qui décrit la structure des tables au niveau *column level*.

Structure au premier niveau

La requête suivante décrit en partie la table Installer. Notez que ça ressemble assez au DESCRIBE (normal car l'instruction a été programmée à l'aide d'une requête analogue).

```
SELECT COLUMN_NAME,DATA_TYPE,ORDINAL_POSITION,COLUMN_DEFAULT,COLUMN_KEY
       FROM  INFORMATION_SCHEMA.COLUMNS
       WHERE TABLE_SCHEMA = 'bdsoutou' AND TABLE_NAME='Installer';
+-------------+-----------+------------------+-------------------+------------+
| COLUMN_NAME | DATA_TYPE | ORDINAL_POSITION | COLUMN_DEFAULT    | COLUMN_KEY |
+-------------+-----------+------------------+-------------------+------------+
| nPoste      | varchar   |                1 | NULL              |            |
| nLog        | varchar   |                2 | NULL              |            |
| numIns      | int       |                3 | NULL              | PRI        |
| dateIns     | timestamp |                4 | CURRENT_TIMESTAMP |            |
| delai       | time      |                5 | NULL              |            |
+-------------+-----------+------------------+-------------------+------------+
```

Remarquez que MySQL utilise :

- COLUMN_NAME pour renseigner le nom de chaque colonne.
- DATA_TYPE pour donner le typeMySQL.
- ORDINAL_POSITION pour renseigner l'ordre des colonnes dans la table (utilisé en cas de SELECT *).
- COLUMN_DEFAULT pour préciser la valeur par défaut de chaque colonne.
- COLUMN_KEY pour donner la composition de la clé primaire.

Extraction des colonnes caractères

La requête suivante décrit en détail les colonnes chaînes de caractères de la table `Installer`.

```
SELECT COLUMN_NAME,DATA_TYPE,CHARACTER_OCTET_LENGTH AS 'Taille max', IS_NULLABLE
       FROM INFORMATION_SCHEMA.COLUMNS
       WHERE TABLE_SCHEMA = 'bdsoutou' AND TABLE_NAME='Installer'
       AND NUMERIC_PRECISION IS NULL
       AND DATA_TYPE NOT IN ('timestamp','time','date');
+------------+----------+-----------+------------+
| COLUMN_NAME | DATA_TYPE | Taille max | IS_NULLABLE |
+------------+----------+-----------+------------+
| nPoste      | varchar   |          7 | YES         |
| nLog        | varchar   |          5 | YES         |
+-------------+-----------+------------+-------------+
```

Vous pouvez noter que MySQL utilise :

- `IS_NULLABLE` pour renseigner le fait qu'une colonne puisse être nulle.
- `CHARACTER_OCTET_LENGTH` pour renseigner la taille des chaînes de caractères pour chaque colonne.

Extraction des colonnes numériques

La requête suivante détaille les colonnes numériques de la table `Installer`.

```
SELECT COLUMN_NAME,DATA_TYPE, NUMERIC_PRECISION AS 'Taille max',
       NUMERIC_SCALE AS 'Précision'
       FROM INFORMATION_SCHEMA.COLUMNS
       WHERE TABLE_SCHEMA = 'bdsoutou' AND TABLE_NAME='Installer'
       AND CHARACTER_MAXIMUM_LENGTH IS NULL
       AND DATA_TYPE NOT IN ('timestamp','time','date');
+-------------+-----------+------------+-----------+
| COLUMN_NAME | DATA_TYPE | Taille max | Précision |
+-------------+-----------+------------+-----------+
| numIns      | int       |         10 |         0 |
+-------------+-----------+------------+-----------+
```

Vous pouvez notez que MySQL utilise :

- `NUMERIC_PRECISION` pour renseigner la taille des numériques pour chaque colonne.
- `NUMERIC_SCALE` pour renseigner la précision des numériques.

Extraction des colonnes date-heure

La requête suivante extrait toutes les colonnes de type date-heure de la table `Installer`.

```
SELECT COLUMN_NAME,DATA_TYPE,COLUMN_DEFAULT
       FROM INFORMATION_SCHEMA.COLUMNS
       WHERE TABLE_SCHEMA = 'bdsoutou' AND TABLE_NAME='Installer'
       AND CHARACTER_MAXIMUM_LENGTH IS NULL
       AND DATA_TYPE IN ('timestamp','time','date');
+-------------+-----------+-------------------+
| COLUMN_NAME | DATA_TYPE | COLUMN_DEFAULT    |
+-------------+-----------+-------------------+
| dateIns     | timestamp | CURRENT_TIMESTAMP |
| delai       | time      | NULL              |
+-------------+-----------+-------------------+
```

Citons, pour en terminer avec cette vue, les colonnes :

- `CHARACTER_SET_NAME` et `COLLATION_NAME` qui renseignent sur le jeu de caractères pour chaque colonne de la table.
- `COLUMN_COMMENT` qui renseigne sur les éventuels commentaires sur chaque colonne.

Les collations et jeux de caractères

Les colonnes `TABLE_COLLATION`, `CHARACTER_SET_NAME` et `COLLATION_NAME` présentes dans différentes vues permettent de retrouver les collations utilisées au niveau des tables et des colonnes.

Collations des tables

La requête suivante détaille le jeu de caractères et la collation associée à une table en particulier.

```
mysql> SELECT c.CHARACTER_SET_NAME, c.COLLATION_NAME
       FROM   INFORMATION_SCHEMA.TABLES t,
         INFORMATION_SCHEMA.COLLATION_CHARACTER_SET_APPLICABILITY c
       WHERE  c.COLLATION_NAME = t.TABLE_COLLATION
         AND  t.TABLE_SCHEMA = "bdutil"
         AND  t.TABLE_NAME = "compagnie";


+--------------------+----------------+
| CHARACTER_SET_NAME | COLLATION_NAME |
+--------------------+----------------+
| latin1             | latin1_bin     |
+--------------------+----------------+
```

Collations des colonnes

La requête suivante détaille le jeu de caractères et la collation associée à chaque colonne d’une table en particulier.

```
mysql> SELECT COLUMN_NAME,CHARACTER_SET_NAME,CONSTRAINT_TYPE
       FROM   INFORMATION_SCHEMA.COLUMNS
       WHERE  TABLE_SCHEMA = "bdutil"
         AND  TABLE_NAME = "compagnie";
+-------------+--------------------+------------------+
| COLUMN_NAME | CHARACTER_SET_NAME | COLLATION_NAME    |
+-------------+--------------------+------------------+
| comp        | latin1             | latin1_bin        |
| nomComp     | latin1             | latin1_general_cs |
| nrue        | NULL               | NULL              |
| rue         | latin1             | latin1_bin        |
| ville       | latin1             | latin1_general_ci |
+-------------+--------------------+------------------+
```

Recherche des contraintes d'une table

La vue TABLE_CONSTRAINTS décrit la nature des contraintes. La requête suivante détaille les contraintes contenues dans la table Installer de la base bdsoutou.

```
SELECT CONSTRAINT_SCHEMA,CONSTRAINT_NAME,CONSTRAINT_TYPE
       FROM INFORMATION_SCHEMA.TABLE_CONSTRAINTS
       WHERE TABLE_SCHEMA = 'bdsoutou' AND TABLE_NAME='Installer';
+------------------+---------------------------+-----------------+
| CONSTRAINT_SCHEMA | CONSTRAINT_NAME           | CONSTRAINT_TYPE |
+------------------+---------------------------+-----------------+
| bdsoutou          | PRIMARY                   | PRIMARY KEY     |
| bdsoutou          | un_installation           | UNIQUE          |
| bdsoutou          | fk_Installer_nLog_Logiciel | FOREIGN KEY     |
| bdsoutou          | fk_Installer_nPoste_Poste  | FOREIGN KEY     |
+------------------+---------------------------+-----------------+
```

MySQL utilise :

- CONSTRAINT_SCHEMA pour indiquer la base de données qui contient la contrainte (qui peut être située dans une autre base de données que la table elle-même).
- CONSTRAINT_NAME pour renseigner le nom de la contrainte. Notez ici que MySQL n'extrait pas le nom de ma contrainte (pk_Installer), sans doute est-ce le fait qu'elle est déclarée également AUTO_INCREMENT.
- CONSTRAINT_TYPE pour renseigner le type de chaque contrainte.

La valeur CHECK dans la colonne CONSTRAINT_TYPE n'est pas encore prise en charge. Vous pouvez toutefois créer des tables avec des contraintes CHECK ; rien ne se passera si vous insérez des données non valides et le dictionnaire restera cohérent en n'extrayant pas ces informations.

Composition des contraintes d’une table

La vue `KEY_COLUMN_USAGE` décrit la composition des contraintes.

La requête suivante permet d’extraire la composition des contraintes de la table `Installer` dans la base `bdsoutou`, et en particulier celle de l’unicité du couple (`nPoste`,`nLog`).

```
SELECT CONSTRAINT_NAME, COLUMN_NAME, ORDINAL_POSITION AS 'Position',
       POSITION_IN_UNIQUE_CONSTRAINT AS 'Position index'
       FROM INFORMATION_SCHEMA.KEY_COLUMN_USAGE
       WHERE TABLE_SCHEMA = 'bdsoutou' AND TABLE_NAME='Installer';
+---------------------------+-------------+----------+---------------+
| CONSTRAINT_NAME           | COLUMN_NAME | Position | Position index |
+---------------------------+-------------+----------+---------------+
| PRIMARY                   | numIns      |        1 |           NULL |
| un_installation           | nPoste      |        1 |           NULL |
| un_installation           | nLog        |        2 |           NULL |
| fk_Installer_nLog_Logiciel | nLog        |        1 |              1 |
| fk_Installer_nPoste_Poste  | nPoste      |        1 |              1 |
+---------------------------+-------------+----------+---------------+
```

MySQL utilise :

- `ORDINAL_POSITION` qui indique la position de la colonne dans la contrainte (débutant à 1).
- `POSITION_IN_UNIQUE_CONSTRAINT` est évaluée à `NULL` pour les index (unique et clé primaire). Pour les clés étrangères composites, elle indique la position de la colonne dans la contrainte.

Détails des contraintes référentielles

Cette même vue permet également de retrouver la nature de la référence pour chaque clé étrangère.

```
SELECT CONSTRAINT_NAME, COLUMN_NAME AS 'Cle',
       REFERENCED_TABLE_SCHEMA AS 'Base cible',
       REFERENCED_TABLE_NAME AS 'Table pere',
       REFERENCED_COLUMN_NAME  AS 'Col pere'
       FROM INFORMATION_SCHEMA.KEY_COLUMN_USAGE
       WHERE TABLE_SCHEMA = 'bdsoutou' AND TABLE_NAME='Installer'
       AND REFERENCED_TABLE_SCHEMA IS NOT NULL;
+---------------------------+--------+------------+------------+----------+
| CONSTRAINT_NAME           | Cle    | Base cible | Table pere | Col pere |
+---------------------------+--------+------------+------------+----------+
| fk_Installer_nLog_Logiciel | nLog   | bdsoutou   | logiciel   | nLog     |
| fk_Installer_nPoste_Poste  | nPoste | bdsoutou   | poste      | nPoste   |
+---------------------------+--------+------------+------------+----------+
```

MySQL utilise :

- REFERENCED_TABLE_SCHEMA qui indique la base de données hébergeant la table « père ». Ici la base données contient les tables « fils » et « pères », mais il se peut que ces tables soient dans deux bases distinctes.
- REFERENCED_TABLE_NAME qui indique le nom de la table « père ».
- REFERENCED_COLUMN_NAME qui indique le nom de la colonne référencée dans la table « père ». Ici, les noms des colonnes des tables « fils » et « père » sont identiques, mais il se peut qu'ils diffèrent.

Depuis la version 5.1, la vue REFERENTIAL_CONSTRAINTS permet de connaître les options relatives à chaque clé étrangère (en modifications et suppressions). Les colonnes UPDATE_RULE et DELETE_RULE désignent respectivement les options ON UPDATE et ON DELETE d'une contrainte référentielle.

La requête suivante interroge cette vue à propos des clés étrangères de la table Installer située dans la base bdsoutou. Les deux contraintes référentielles de cette table ont été créées sans option, c'est donc la valeur RESTRICT qui est associée aux clés étrangères (les autres valeurs possibles sont CASCADE, SET NULL, SET DEFAULT et NO ACTION).

```
SELECT CONSTRAINT_NAME,UPDATE_RULE,DELETE_RULE
       REFERENCED_TABLE_NAME "Table cible"
  FROM  INFORMATION_SCHEMA.REFERENTIAL_CONSTRAINTS
  WHERE TABLE_NAME       = 'Installer'
  AND   CONSTRAINT_SCHEMA = 'bdsoutou';


+---------------------------+-------------+-------------+-------------+
| CONSTRAINT_NAME           | UPDATE_RULE | DELETE_RULE | Table cible |
+---------------------------+-------------+-------------+-------------+
| fk_Installer_nLog_Logiciel | RESTRICT    | RESTRICT    | logiciel    |
| fk_Installer_nPoste_Poste  | RESTRICT    | RESTRICT    | poste       |
+---------------------------+-------------+-------------+-------------+
```

Recherche du code source d'un sous-programme

La vue ROUTINES décrit la composition des sous-programmes (procédures et fonctions cataloguées).

Supposons que la procédure sp1(par_1 CHAR(5), par_2 SMALLINT) et la fonction sp2(par_3 INT) RETURNS CHAR(3) soient compilées dans la base test. La requête suivante permet d'extraire le code source de ces sous-programmes.

```
SELECT ROUTINE_NAME,ROUTINE_TYPE,ROUTINE_DEFINITION
       FROM  INFORMATION_SCHEMA.ROUTINES
       WHERE ROUTINE_SCHEMA = 'test';
+-------------+-------------+------------------------------------------------+
```

```
| ROUTINE_NAME | ROUTINE_TYPE | ROUTINE_DEFINITION                                |
+--------------+--------------+---------------------------------------------------+
| sp1          | PROCEDURE    | BEGIN
                                DECLARE v_brevet CHAR(6);
                                        SET v_brevet := 'PROC';
                                END                                               |
| sp2          | FUNCTION     | BEGIN
                                DECLARE v_brevet CHAR(3);
                                        SET v_brevet := 'FCT';
                                        RETURN v_brevet ;
                                END                                               |
+--------------+--------------+---------------------------------------------------+
```

MySQL utilise :

- `ROUTINE_SCHEMA` qui indique le nom de la base hébergeant le sous-programme.
- `ROUTINE_NAME` qui indique le nom du sous-programme.
- `ROUTINE_TYPE` qui indique la nature du sous-programme (fonction ou procédure).
- `ROUTINE_DEFINITION` qui liste le code MySQL du sous-programme.

Citons, pour en terminer avec cette vue, les colonnes :

- `SECURITY_TYPE` qui renseigne sur les privilèges associés à la vue lors de son exécution (soit les privilèges de l'utilisateur créateur : *definer* ; soit ceux de l'utilisateur qui lance l'exécution : *invoker*).
- `CREATED` et `LAST_ALTERED` pour stocker la date de création du sous-programme et l'instant de la dernière compilation.
- `DEFINER` qui indique l'identité de l'utilisateur qui a créé le sous-programme.
- `ROUTINE_COMMENT` qui stocke un éventuel commentaire relatif au sous-programme (initialisé lors de la compilation).

Paramètres des sous-programmes stockés

Depuis la version 5.5.3, la vue `PARAMETERS` détaille les paramètres des procédures et fonctions cataloguées. La requête suivante extrait les paramètres des deux sous-programmes de la base `test`.

```
SELECT SPECIFIC_NAME,ORDINAL_POSITION,
       PARAMETER_MODE,PARAMETER_NAME
  FROM INFORMATION_SCHEMA.PARAMETERS
  WHERE SPECIFIC_SCHEMA='test';
+---------------+------------------+---------------+---------------+
| SPECIFIC_NAME | ORDINAL_POSITION | PARAMETER_MODE | PARAMETER_NAME |
+---------------+------------------+---------------+---------------+
| sp1           |                1 | IN             | par_1          |
```

```
| sp1           |               2 | IN            | par_2          |
| sp2           |               0 | NULL          | NULL           |
| sp2           |               1 | IN            | par_3          |
+---------------+-----------------+---------------+----------------+
```

MySQL utilise :

- La colonne `ORDINAL_POSITION` désigne la position des paramètres (1, 2, 3…). La valeur 0 désigne le paramètre de retour d'une fonction.
- La colonne `PARAMETER_MODE` détermine le type du paramètre (`IN` en entrée, `OUT` en sortie, `INOUT` les deux ou `NULL` pour le paramètre de retour d'une fonction).

D'autres colonnes existent et précisent la nature de chaque paramètre, citons `DATA_TYPE`, `CHARACTER_MAXIMUM_LENGTH`, `CHARACTER_OCTET_LENGTH`, `NUMERIC_PRECISION`, `NUMERIC_SCALE`, `CHARACTER_SET_NAME`, `COLLATION_NAME` et `DTD_IDENTIFIER`.

Privilèges des utilisateurs d'une base de données

On retrouve les différents niveaux de privilèges étudiés en début de chapitre.

Au niveau global

La vue `USER_PRIVILEGES` liste les privilèges des accès utilisateurs au niveau global (les données viennent de la table `mysql.user`). La requête suivante extrait les privilèges de `Paul` et de `Jules` (en accès distant ou en local). Non, vous ne rêvez pas, trois simples quotes sont nécessaires pour tester la valeur de la colonne `GRANTEE`.

```
SELECT GRANTEE,PRIVILEGE_TYPE,IS_GRANTABLE
       FROM  INFORMATION_SCHEMA.USER_PRIVILEGES
       WHERE GRANTEE LIKE '''Paul%' OR GRANTEE LIKE '''Jules%';
+-----------------------+---------------+--------------+
| GRANTEE               | PRIVILEGE_TYPE | IS_GRANTABLE |
+-----------------------+---------------+--------------+
| 'Paul'@'localhost'     | USAGE          | NO           |
| 'Jules'@'localhost'    | USAGE          | NO           |
| 'Paul'@'192.168.4.173' | SELECT         | NO           |
| 'Paul'@'192.168.4.173' | CREATE         | NO           |
+-----------------------+---------------+--------------+
```

MySQL utilise :

- `GRANTEE` qui indique le nom de l'accès utilisateur.
- `PRIVILEGE_TYPE` qui indique le type du privilège.
- `IS_GRANTABLE` qui renseigne sur la possibilité que l'accès utilisateur puisse

retransmettre le privilège acquis (reçu avec `WITH GRANT OPTION`).

Au niveau database

La vue `SCHEMA_PRIVILEGES` possède la même structure que la précédente, en ajoutant la colonne `TABLE_SCHEMA`. Celle-ci donne le nom de la base de données concernée par les privilèges des accès utilisateurs (les données viennent de la table `db.user`). La requête suivante extrait les privilèges au niveau *database* de `Paul`, en accès distant ou local.

```
SELECT TABLE_SCHEMA,GRANTEE,PRIVILEGE_TYPE,IS_GRANTABLE
       FROM  INFORMATION_SCHEMA.SCHEMA_PRIVILEGES
       WHERE GRANTEE LIKE '''Paul%';
+--------------+-----------------------+----------------+--------------+
| TABLE_SCHEMA | GRANTEE               | PRIVILEGE_TYPE | IS_GRANTABLE |
+--------------+-----------------------+----------------+--------------+
| bdpaul       | 'Paul'@'192.168.4.173' | SELECT         | NO           |
| bdpaul       | 'Paul'@'localhost'     | DROP           | NO           |
| bdpaul       | 'Paul'@'localhost'     | CREATE ROUTINE | NO           |
+--------------+-----------------------+----------------+--------------+
```

Au niveau table

La vue `TABLE_PRIVILEGES` possède la même structure que la précédente, en ajoutant la colonne `TABLE_NAME`. Celle-ci donne le nom de la table concernée par les privilèges des accès utilisateurs (les données viennent de la table `mysql.tables_priv`). La requête suivante extrait les privilèges au niveau *table* de `Paul`, en accès distant ou local.

```
SELECT CONCAT(TABLE_SCHEMA,'.',TABLE_NAME) AS 'Base.Table',
       GRANTEE,PRIVILEGE_TYPE AS 'Privilege'
       FROM  INFORMATION_SCHEMA.TABLE_PRIVILEGES
       WHERE GRANTEE LIKE '''Paul%';
+-----------------------------------+-----------------------+-----------+
| Base.Table                        | GRANTEE               | Privilege |
+-----------------------------------+-----------------------+-----------+
| bdsoutou.VueDesCompagniesJoursFeries | 'Paul'@'localhost'     | SELECT    |
| bdpaul.Livre                      | 'Paul'@'192.168.4.173' | SELECT    |
| bdpaul.Livre                      | 'Paul'@'localhost'     | SELECT    |
| bdpaul.Livre                      | 'Paul'@'localhost'     | DELETE    |
+-----------------------------------+-----------------------+-----------+
```

Au niveau column

La vue `COLUMN_PRIVILEGES` possède la même structure que la précédente, en ajoutant la colonne `COLUMN_NAME`. Celle-ci précise le nom de la colonne concernée par les privilèges des accès utilisateurs (les données viennent de la table `mysql columns_priv`). La requête suivante extrait les privilèges au niveau *column* de `Paul`,

en accès distant ou local sur la base `bdpaul`.

```
SELECT CONCAT(TABLE_NAME,'.',COLUMN_NAME) AS 'Table.colonne',
       GRANTEE,PRIVILEGE_TYPE AS 'Privilege'
       FROM  INFORMATION_SCHEMA.COLUMN_PRIVILEGES
       WHERE  TABLE_SCHEMA='bdpaul';
+--------------+--------------------+-----------+
| Table.colonne | GRANTEE            | Privilege |
+--------------+--------------------+-----------+
| Livre.ISBN    | 'Paul'@'localhost' | UPDATE    |
+--------------+--------------------+-----------+
```

Au niveau routine

Se reporter au niveau *database*.

Commande `SHOW`

La commande `SHOW` permet d'extraire facilement des informations provenant du dictionnaire des données. Elle est bien sûr, à l'inverse, plus limitée que l'écriture d'une requête `SELECT` – qui pourra toujours extraire les mêmes informations, mais en interrogeant les vues adéquates. La copie d'écran suivante illustre la commande `SHOW TABLES` qui restitue une réponse à la question : « Quelles sont les tables et les vues présentes dans la base de données en cours d'utilisation ? ».

Figure 5-14 *Exemple de `SHOW` pour lister les tables d'un schéma*

Le tableau suivant décrit quelques exemples qui vous seront peut-être utiles.

Tableau 5-29 Exemples de `SHOW`

Commande	Résultat
`SHOW COLUMNS FROM Installer FROM bdsoutou LIKE 'n%';`	Liste des colonnes dont le nom commence par '`n`' dans la table `Installer` de la base `bdsoutou`.
`SHOW CREATE DATABASE bdsoutou;`	Options de création de la base `bdsoutou`.
`SHOW CREATE TABLE bdsoutou.Installer;`	Description totale de l'instruction permettant de créer la table `Installer` de la base `bdsoutou`.
`SHOW DATABASES;`	Liste des bases présentes sur le serveur.
`SHOW ENGINES;`	Liste des moteurs de stockage utilisables sur le serveur.
`SHOW ERRORS;`	Libellé de l'erreur SQL courante.
`SHOW GRANTS FOR 'Paul'@'localhost';`	Pour un accès utilisateur, liste de ses privilèges aux niveaux *global*, *database*, *column* et *routine*.
`SHOW INDEX FROM Installer FROM bdsoutou;`	Description des index de la table `Installer` de la base `bdsoutou`.
`SHOW PRIVILEGES;`	Liste de tous les privilèges possibles.
`SHOW TABLE STATUS FROM bdsoutou LIKE 'S%';`	Caractéristiques physiques des tables dont le nom commence par '`S`' dans la base `bdsoutou`.
`SHOW TABLES FROM mysql;`	Liste des tables de la base `mysql`.
`SHOW TRIGGERS;`	Caractéristiques des déclencheurs présents sur le serveur.

Exercices

Les objectifs de ces exercices sont :

- de créer des vues monotables et multitables ;
- d'insérer des enregistrements dans des vues ;
- d'effectuer une mise à jour conditionnée via une vue.

Exercice 5.1

Vues monotables

Vues sans contraintes

Écrire le script `vues.sql`, permettant de créer :

La vue `LogicielsUnix` qui contient tous les logiciels de type 'UNIX' (toutes les colonnes sont conservées). Vérifier la structure et le contenu de la vue (`DESCRIBE` et `SELECT`).

La vue `Poste_0` de structure (`nPos0`, `nomPoste0`, `nSalle0`, `TypePoste0`, `indIP`, `ad0`) qui contient tous les postes du rez-de-chaussée (`etage=0` au niveau de la table `Segment`). Faire une jointure procédurale, sinon la vue sera considérée comme une vue multitable. Vérifier la structure et le contenu de la vue.

Insérer deux nouveaux postes dans la vue tels qu'un poste soit connecté au segment du rez-de-chaussée, et l'autre à un segment d'un autre étage. Vérifier le contenu de la vue et celui de la table. Conclusion ?

Supprimer ces deux enregistrements de la table `Poste`.

Résoudre une requête complexe

Créer la vue `SallePrix` de structure (`nSalle`, `nomSalle`, `nbPoste`, `prixLocation`) qui contient les salles et leur prix de location pour une journée (en fonction du nombre de postes). Le montant de la location d'une salle à la journée sera d'abord calculé sur la base de 100 € par poste. Servez-vous de l'expression `100*nbPoste` dans la requête de définition.

Vérifier le contenu de la vue, puis afficher les salles dont le prix de location dépasse 150 €.

Ajouter la colonne `tarif` de type `SMALLINT(4)` à la table `Types`. Mettre à jour cette table de manière à insérer les valeurs suivantes :

Tableau 5-30 Tarifs des postes

Type du poste	Tarif en €
TX	50
PCWS	100

PCNT	120
UNIX	200
NC	80
BeOS	400

Créer la vue `SalleIntermediaire` de structure (`nSalle`, `typePoste`, `nombre`, `tarif`), de telle sorte que le contenu de la vue reflète le tarif ajusté des salles, en fonction du nombre et du type des postes de travail. Il s'agit de grouper par salle, type et tarif (tout en faisant une jointure avec la table `Types` pour les tarifs), et de compter le nombre de postes pour avoir le résultat suivant :

```
+--------+-----------+--------+-------+
| nSalle | typePoste | nombre | tarif |
+--------+-----------+--------+-------+
| s01    | TX        |      2 |    50 |
| s01    | UNIX      |      2 |   200 |
| s02    | PCWS      |      2 |   100 |
| s03    | TX        |      1 |    50 |
| ...                                 |
```

À partir de la vue `SalleIntermediaire`, créer la vue `SallePrixTotal`(`nSalle`, `PrixReel`) qui reflète le prix réel de chaque salle (par exemple, la s01 sera facturée 2*50 + 1*200 = 300 €). Vérifier le contenu de cette vue.

Afficher les salles les plus économiques à la location.

Vues avec contraintes

Remplacer la vue `Poste0` en rajoutant l'option de contrôle (`CHECK OPTION`). Tenter d'insérer un poste appartenant à un étage différent du rez-de-chaussée.

Créer la vue `Installer0` de structure (`nPoste`, `nLog`, `dateIns`) ne permettant de travailler qu'avec les postes du rez-de-chaussée, tout en interdisant l'installation d'un logiciel de type 'PCNT'. Tenter d'insérer deux postes, dans cette vue, ne correspondant pas à ces deux contraintes : un poste d'un étage, puis un logiciel de type 'PCNT'. Insérer l'enregistrement 'p6', 'log2' qui doit passer à travers la vue.

Exercice 5.2

Vue multitable

Créer la vue `SallePoste` de structure (`nomSalle`, `nomPoste`, `adrIP`, `nomTypePoste`) permettant d'extraire toutes les installations sous la forme suivante :

```
SELECT * FROM SallePoste;
+----------+----------+--------------+-------------------+
| nomSalle | nomPoste | adrIP        | nomTypePoste      |
+----------+----------+--------------+-------------------+
| Salle 1  | Poste 1  | 130.120.80.01 | Terminal X-Window |
| Salle 1  | Poste 2  | 130.120.80.02 | Système Unix      |
| Salle 1  | Poste 3  | 130.120.80.03 | Terminal X-Window |
| ...                                                    |
```

1. Je conserve le vocable de « vue » pour être plus près de la réalité. Cependant, parler de *table* ou de *vue* est équivalent, puisqu'elles sont interrogeables de la même manière : par `SELECT`.

Partie II

Programmation procédurale

Chapitre 6

Bases du langage de programmation

Ce chapitre décrit les caractéristiques générales du langage procédural de programmation de MySQL :

- structure d'un programme ;
- déclaration et affectation de variables ;
- structures de contrôle (*si*, *tant que*, *répéter*, *pour*) ;
- mécanismes d'interaction avec la base ;
- gestion des transactions.

Généralités

Les structures de contrôle habituelles d'un langage (`IF`, `WHILE`…) ne font pas partie intégrante de la norme SQL. Elles apparaissent dans une sous-partie optionnelle de la norme (ISO/IEC 9075-5:1996. *Flow-control statements*). MySQL les prend en compte.

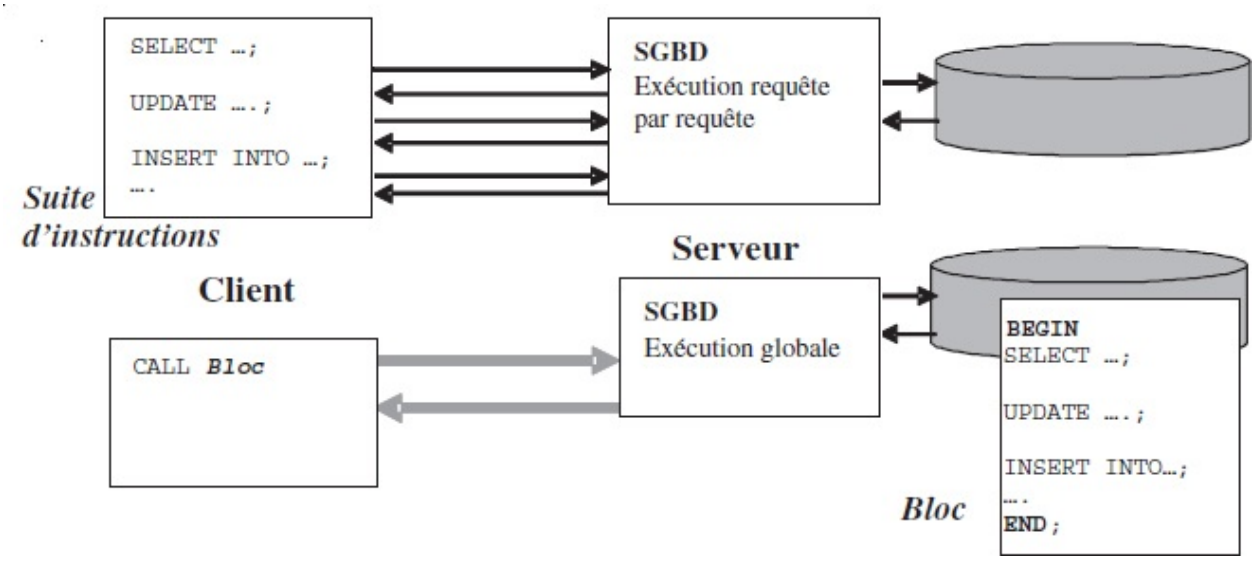
Le langage procédural de MySQL est une extension de SQL, car il permet de faire cohabiter les habituelles structures de contrôle (*si*, *pour* et *tant que* pour les plus connues) avec des instructions SQL (principalement `SELECT`, `INSERT`, `UPDATE` et `DELETE`).

Environnement client-serveur

Dans un environnement client-serveur, chaque instruction SQL donne lieu à l'envoi d'un message du client vers le serveur suivi de la réponse du serveur vers le client. Il est préférable de travailler avec un sous-programme (qui sera stocké, en fait, côté serveur) plutôt qu'avec une suite d'instructions SQL susceptibles d'encombrer le trafic réseau. En effet, un bloc donne lieu à un seul

échange sur le réseau entre le client et le serveur. Les résultats intermédiaires sont traités côté serveur et seul le résultat final est retourné au client.

Figure 6-1 *Trafic sur le réseau d'instructions SQL*



Avantages

Les principaux avantages d'utiliser des sous-programmes (procédures ou fonctions cataloguées qui sont stockées côté serveur) sont :

- La modularité : un sous-programme peut être composé d'autres blocs d'instructions. Un sous-programme peut aussi être réutilisable, car il peut être appelé par un autre.
- La portabilité : un sous-programme est indépendant du système d'exploitation qui héberge le serveur MySQL. En changeant de système, les applicatifs n'ont pas à être modifiés.
- L'intégration avec les données des tables : on retrouvera avec ce langage procédural tous les types de données et d'instructions disponibles sous MySQL, des mécanismes pour parcourir des résultats de requêtes (curseurs), pour traiter des erreurs (*handlers*) et pour programmer des transactions (`COMMIT`, `ROLLBACK`, `SAVEPOINT`).
- La sécurité, car les sous-programmes s'exécutent dans un environnement *a priori* sécurisé (SGBD) où il est plus facile de garder la maîtrise sur les ordres SQL exécutés et donc sur les agissements des utilisateurs.

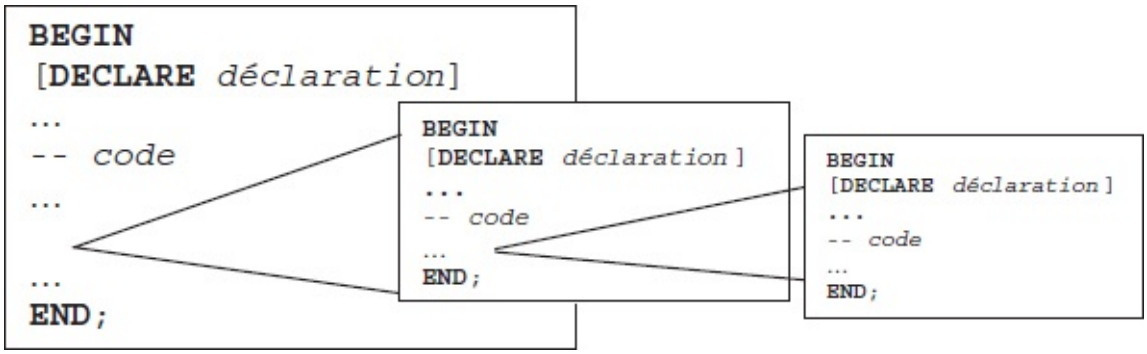
Structure d’un bloc

Un bloc d’instructions est composé de :

- `BEGIN` (section obligatoire) contient le code incluant ou non des directives SQL se terminant par le symbole « ; » ;
- `DECLARE` (directive optionnelle) déclare une variable, un curseur, une exception, etc. ;
- `END` ferme le bloc.

Un bloc peut être imbriqué dans un autre bloc. Pour tester un bloc, nous verrons dans la section « Tests des exemples », qu’il faut l’inclure dans une procédure cataloguée. MySQL ne prend pas encore en charge les procédures anonymes (sans nom).

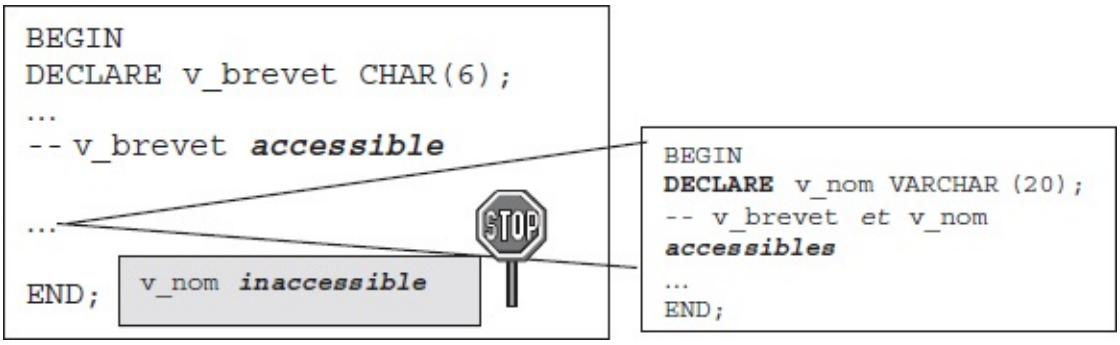
Figure 6-2 *Structure d’un bloc d’instructions MySQL*



Portée des objets

La portée d’un objet (variable, curseur ou exception) est la zone du programme qui peut y accéder. Un objet déclaré dans un bloc est accessible dans les sous-blocs. En revanche, un objet déclaré dans un sous-bloc n’est pas visible du bloc supérieur (principe des accolades des langages C et Java).

Figure 6-3 *Visibilité des objets*



Casse et lisibilité

Comme SQL, les sous-programmes sont capables d'interpréter les caractères alphanumériques du jeu de caractères sélectionné. Aucun objet manipulé par programme n'est sensible à la casse (*not case sensitive*). Ainsi `numeroBrevet` et `NumeroBREVET` désignent le même identificateur (tout est traduit en minuscules au niveau du dictionnaire des données). Les règles d'écriture classiques concernant l'indentation et les espaces entre variables, mots-clés et instructions doivent être respectées dans un souci de lisibilité.

Tableau 6-1 Lisibilité du code

Peu lisible	C'est mieux

`IF x>y THEN SET max:=x;ELSE SET max:=y;END IF;`	`IF x>y THEN`
	`  SET max:=x;`
	`ELSE`
	`  SET max:=y;`
	`END IF;`

Identificateurs

Avant de parler des différents types de variables MySQL, décrivons comment il est possible de nommer les objets des sous-programmes. Un identificateur commence par une lettre (ou un chiffre). Un identificateur n'est pas limité en nombre de caractères. Les autres signes pourtant connus du langage sont interdits, comme le montre le tableau suivant :

Tableau 6-2 Identificateurs

Autorisés	Interdits
`t2`	`moi&toi` (symbole « & »)

`code_brevet`
`2nombresMysql`
`_t`
	`debit-credit` (symbole « - »)
	`on/off` (symbole « / »)
	`code brevet` (symbole espace)

Commentaires

MySQL prend en charge deux types de commentaires : monolignes, commençant au symbole « `--` » et finissant à la fin de la ligne ; et multilignes, commençant par « `/*` » et finissant par « `*/` ». Le tableau suivant décrit quelques exemples :

Tableau 6-3 Commentaires

Sur une ligne	Sur plusieurs lignes

`-- Lecture de la table Pilote`
`SELECT  nbHVol INTO v_nbHVol`
`  FROM  Pilote -- Extraction heures de vol`
`  WHERE nom = 'Gratien Viel';`
`SET v_bonus := v_nbHVol*0.15; -- Calcul`
	`/* Lecture de la table Pilote */`
	`SELECT salaire INTO v_salaire`
	`  FROM Pilote`
	`  /* Extraction du salaire`
	`  pour calculer le bonus */`
	`  WHERE nom = 'Thierry Albaric';`
	`/*Calcul*/`
	`SET v_bonus := v_salaire*0.15;`

Variables

Un sous-programme est capable de manipuler des variables qui sont déclarées (et éventuellement initialisées) par la directive `DECLARE`. Ces variables permettent de transmettre des valeurs à des sous-programmes via des paramètres, ou d'afficher des états de sortie sous l'interface. Deux types de variables sont disponibles sous MySQL :

- scalaires : recevant une seule valeur d'un type SQL (ex : colonne d'une table) ;
- externes : définies dans la session et qui peuvent servir de paramètres d'entrée ou de sortie.

Variables scalaires

La déclaration d'une variable scalaire est de la forme suivante :

```
DECLARE nomVariable1[,nomVariable2...] typeMySQL [DEFAULT expression];
```

- `DEFAULT` permet d'initialiser la (ou les) variable(s) – pas forcément à l'aide d'une constante. Le tableau suivant décrit quelques exemples :

Tableau 6-4 Déclarations

Déclarations	Commentaires
`DECLARE v_dateNaissance DATE;`	Déclare la variable sans l'initialiser. Équivalent à `SET v_dateNaissance := NULL;`
`DECLARE v_capacite SMALLINT(4) DEFAULT 999;`	Initialise la variable à 999.
`DECLARE v_trouve BOOLEAN DEFAULT TRUE;`	Initialise la variable à vrai (1).
`DECLARE v_Dans2jours DATE DEFAULT ADDDATE(SYSDATE(),2);`	Initialise la variable à dans 2 jours.

Affectations

Il existe plusieurs possibilités pour affecter une valeur à une variable :

- l'affectation comme on la connaît dans les langages de programmation (`SET variable := expression`). Vous pouvez aussi utiliser le symbole « = », mais il est plus prudent de le réserver à la programmation de conditions ;
- la directive `DEFAULT` ;
- la directive `INTO` d'une requête (`SELECT … INTO variable FROM …`).

Restrictions

Le type tableau (*array*) n'est pas encore présent dans le langage de MySQL. Cela peut être pénalisant quand on désire travailler en interne avec des résultats d'extractions de taille moyenne.

Il est impossible d'utiliser un identificateur dans une expression, s'il n'est pas déclaré au préalable. Ici, la déclaration de la variable `v_maxi` est incorrecte :

```
DECLARE   v_maxi INT DEFAULT 2 * v_mini;
```

```
DECLARE   v_mini INT DEFAULT 15;
```

Comme la plupart des langages récents, les déclarations multiples sont permises. Celle qui suit est juste :

```
DECLARE i, j, k INT;
```

Résolution de noms

Lors des conflits potentiels de noms (variables ou colonnes) dans des instructions SQL (principalement INSERT, UPDATE, DELETE et SELECT), le nom de la variable est prioritairement interprété au détriment de la colonne de la table (de même nom).

Dans l'exemple suivant, l'instruction DELETE supprime tous les pilotes de la table (et non pas seulement le pilote de nom 'Placide Fresnais'), car MySQL considère les deux identificateurs comme étant la même variable, et non pas comme colonne de la table et variable.

```
DECLARE nom VARCHAR(16) DEFAULT 'Placide Fresnais';
DELETE FROM Pilote WHERE nom = nom ;
```

Pour se prémunir de tels effets de bord, une seule solution existe : elle consiste à nommer toutes les variables différemment des colonnes (en utilisant un préfixe, par exemple). Une autre solution serait d'utiliser une étiquette de bloc (*block label*) pour lever d'éventuelles ambiguïtés. Bien qu'il soit possible d'employer des étiquettes de blocs (aussi disponibles pour les structures de contrôle), on ne peut pas encore préfixer des variables pour en distinguer une de même nom entre différents blocs.

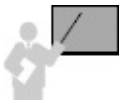
Tableau 6-5 Éviter les ambiguïtés

Préfixer les variables	Étiquette de bloc (préfixe pas opérationnel)

`DECLARE v_nom VARCHAR(16)`
`  DEFAULT 'Placide Fresnais';`
`…`
`DELETE FROM Pilote WHERE nom = v_nom;`
`--ou`
`DELETE FROM Pilote WHERE v_nom = nom;`
	`principal: BEGIN`
	` DECLARE nom VARCHAR(16)`
	`  DEFAULT 'Placide Fresnais';`
	` DELETE  FROM Pilote`
	`  WHERE principal.nom = nom;`
	`END principal;`

Opérateurs

Les opérateurs SQL étudiés au chapitre 4 (logiques, arithmétiques, de concaténation…) sont disponibles au sein d'un sous-programme. Les règles de priorité sont les mêmes que dans le cas de SQL.



L'opérateur `IS NULL` permet de tester une formule avec la valeur `NULL`. Toute expression arithmétique contenant une valeur nulle est évaluée à `NULL`.

Le tableau suivant illustre quelques utilisations possibles d'opérateurs logiques :

Tableau 6-6 Utilisation d'opérateurs

Code MySQL	Commentaires

`DECLARE v_compteur INT(3) DEFAULT 0;`
`DECLARE v_boolean BOOLEAN;`
`DECLARE v_nombre INT(3);`
	Trois déclarations dont une avec initialisation.

`SET v_compteur := v_compteur+1;`	Incrémentation de `v_compteur` (opérateur +)
`SET v_boolean := (v_compteur=v_nombre);`	`v_boolean` reçoit **`NULL`**, car la condition est fausse.
`SET v_boolean := (v_nombre IS NULL);`	`v_boolean` reçoit **`TRUE`** (1 en fait), car la condition est vraie.

Variables de session

Il est possible de passer en paramètres d'entrée d'un bloc des variables externes. Ces variables sont dites de session (*user variables*). Elles n'existent que durant la session. On déclare ces variables en ligne de commande à l'aide du symbole « @ ».

```
SET @var1 = expression1 [, @var2 = expression2] ...
```

Le tableau suivant illustre un exemple de deux variables de session : on extrait le nom et le nombre d'heures de vol d'un pilote (table décrite au début du chapitre 4) augmenté d'un nombre en paramètre. Son numéro de brevet et la durée du vol sont lus au clavier. Ces variables de session ne sont bien sûr pas à déclarer dans le bloc

Tableau 6-7 Variables de session

Code MySQL	Résultat

`delimiter $`
 	
`SET @vs_num = 'PL-4'$`
`SET @vs_hvol = 15$`
`…`
`BEGIN`
`  DECLARE v_nom    CHAR(16);`
`  DECLARE v_nbHVol DECIMAL(7,2);`
`  SELECT nom,nbHVol`
`       INTO v_nom, v_nbHVol FROM Pilote`
`       WHERE brevet = @vs_num;`
`  SET v_nbHVol := v_nbHVol + @vs_hvol;`
`  SELECT v_nom, v_nbHVol;`
`END;`
`$`
	`mysql> CALL sp1()$`
	
	`+------------------+----------+`
	`| v_nom  | v_nbHVol |`
	`+------------------+----------+`
	`| Placide Fresnais |  2465.00 |`
	`+------------------+----------+`

Conventions recommandées

Adoptez les conventions d'écriture suivantes pour que vos programmes MySQL soient plus facilement lisibles et maintenables :

Tableau 6-8 Conventions

Objet	Convention	Exemple
Variable	**v**_*nomVariable*	v_compteur
Constante	**c**_nomConstante	c_pi
Variable de session (globale)	**vs**_*nomVariable*	vs_brevet

Test des exemples

Parce qu'il n'est pas encore possible d'exécuter des blocs anonymes (sous-programme sans nom et qui n'est pas stocké dans la base), vous devez les inclure dans une procédure cataloguée que vous appellerez dans l'interface de commande.

L'exemple suivant extrait le nombre d'heures de vol du pilote de nom 'Placide Fresnais'. Pensez à redéfinir le délimiteur à « $ » (par exemple) pour pouvoir utiliser, dans le bloc, le symbole « ; » pour terminer chaque instruction.

Tableau 6-9 Tester un exemple de bloc

Préfixer les variables	Commentaire
`delimiter $` `SET @vs_nom = 'Placide Fresnais'$`	Déclaration du délimiteur et d'une variable de session.
`DROP PROCEDURE sp1$`	Suppression de la procédure.
`CREATE PROCEDURE sp1()`	Création de la procédure.

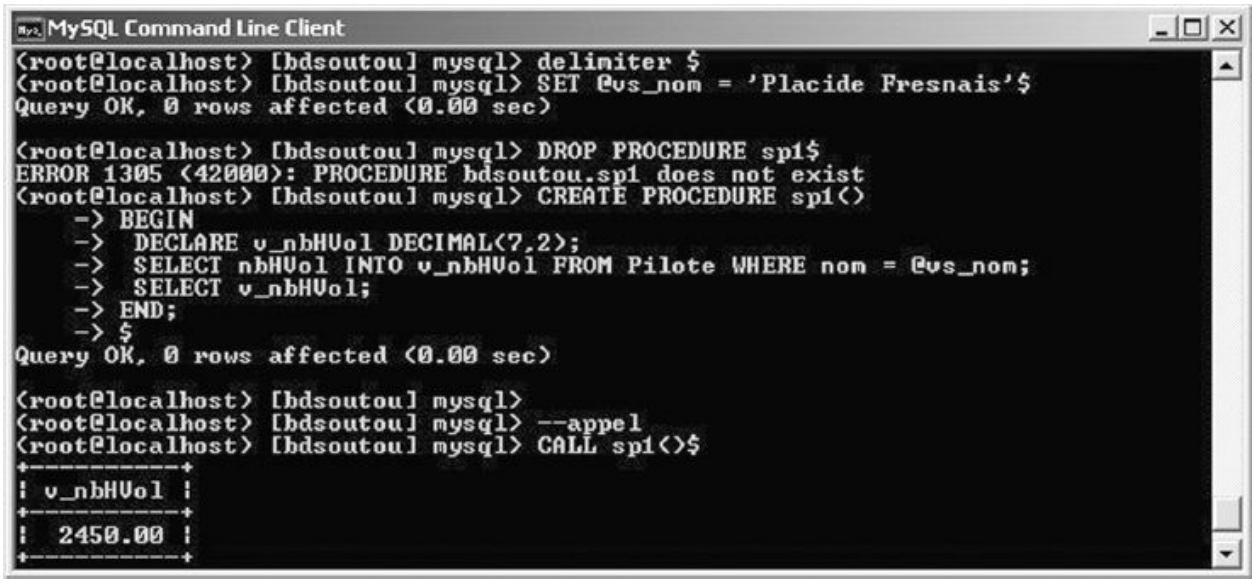
`BEGIN`
`  DECLARE v_nbHVol DECIMAL(7,2);`
`  SELECT nbHVol INTO v_nbHVol`
`  FROM Pilote WHERE nom = @vs_nom;`
	Bloc d'instructions.

`  SELECT v_nbHVol;`
`END;`
`$`
	Trace du résultat. Fin du bloc

`CALL sp1()$`	Appel de la procédure.

Le résultat dans l'interface de commande est le suivant. Allez-y tester vos exemples, maintenant.

Figure 6-4 *Exécution d'un bloc*



Structures de contrôle

En tant que langage procédural, MySQL offre la possibilité de programmer :

- les structures conditionnelles *si* et *cas* (`IF… et CASE`) ;
- des structures répétitives *tant que*, *répéter* et *boucle sans fin* (`WHILE`, `REPEAT` et

LOOP).

Pas de structure FOR pour l'instant. Deux directives supplémentaires qui sont toutefois à utiliser avec modération : LEAVE qui sort d'une boucle (ou d'un bloc étiqueté) et ITERATE qui force le programme à refaire un tour de boucle depuis le début.

Structures conditionnelles

MySQL propose deux structures pour programmer des actions conditionnées : la structure IF et la structure CASE.

Trois formes de IF

Suivant les tests à programmer, on peut distinguer trois formes de structure IF : IF-THEN (*si-alors*), IF-THEN-ELSE (avec le *sinon* à programmer), et IF-THEN-ELSEIF (imbrications de conditions).

Le tableau suivant donne l'écriture des différentes structures conditionnelles IF. Notez « END IF » en fin de structure, et non pas « ENDIF ». L'exemple affiche un message différent selon la nature du numéro de téléphone contenu dans la variable v_telephone.

Tableau 6-10 Structures IF

IF-THEN	IF-THEN-ELSE	IF-THEN-ELSEIF

IF *condition* **THEN**
  instructions;
END IF;
	IF *condition* **THEN**
	  instructions;
	ELSE
	  instructions;
	END IF;
		IF *condition1* **THEN**
		  instructions;
		ELSEIF *condition2* **THEN**
		  instructions2;
		  ELSE
		  instructions3;
		END IF;

```
BEGIN
  DECLARE v_telephone CHAR(14)
         DEFAULT '06-76-85-14-89';
  IF SUBSTR(v_telephone,1,2)='06' THEN
     SELECT "C'est un portable";
  ELSE
     SELECT "C'est un fixe...";
  END IF;
END;
```

Conditions booléennes

Les tableaux suivants précisent le résultat d'opérateurs logiques qui mettent en jeu des variables booléennes pouvant prendre trois valeurs (TRUE, FALSE, NULL). Bien sûr, en l'absence d'un vrai type booléen, MySQL représente TRUE avec 1, et FALSE avec 0. Il est à noter que la négation de NULL (NOT NULL) renvoie une valeur nulle.

Tableau 6-11 Opérateur AND

AND	TRUE	FALSE	NULL
TRUE	TRUE	FALSE	NULL
FALSE	FALSE	FALSE	FALSE
NULL	NULL	FALSE	NULL

Tableau 6-12 Opérateur OR

OR	TRUE	FALSE	NULL
TRUE	TRUE	TRUE	TRUE
FALSE	TRUE	FALSE	NULL
NULL	TRUE	NULL	NULL

Structure CASE

Comme l'instruction IF, la structure CASE permet d'exécuter une séquence d'instructions en fonction de différentes conditions. La structure CASE est utile lorsqu'il faut évaluer une même expression et proposer plusieurs traitements pour diverses conditions.

Selon la nature de l'expression et des conditions, une des deux écritures suivantes peut être utilisée :

Tableau 6-13 Structures CASE

CASE	***searched*** CASE

CASE *variable*
  WHEN *expr1* THEN *instructions1*;
  WHEN *expr2* THEN *instructions2*;
  …
  WHEN *exprN* THEN *instructionsN*;
  [ELSE *instructionsN+1*;]
END CASE;
	CASE
	  WHEN *condition1* THEN *instructions1*;
	  WHEN *condition2* THEN *instructions2*;
	  …
	  WHEN *conditionN* THEN *instructionsN*;
	  [ELSE *instructionsN+1*;]
	END CASE;

Le tableau suivant nous livre l'écriture avec IF d'une programmation qu'il est plus rationnel d'effectuer avec une structure CASE (de type *searched*) :

Tableau 6-14 Différentes programmations

IF	CASE
	BEGIN DECLARE v_mention CHAR(2); DECLARE v_note DECIMAL(4,2) DEFAULT 9.8; ...

IF v_note >= 16 THEN
SET v_mention := 'TB';
ELSEIF v_note >= 14 THEN
SET v_mention := 'B';
ELSEIF v_note >= 12 THEN
SET v_mention := 'AB';
ELSEIF v_note >= 10 THEN
SET v_mention := 'P';
ELSE
SET v_mention := 'R';
END IF;
...
	CASE
	WHEN v_note >= 16 THEN SET v_mention := 'TB';
	WHEN v_note >= 14 THEN SET v_mention := 'B';
	WHEN v_note >= 12 THEN SET v_mention := 'AB';
	WHEN v_note >= 10 THEN SET v_mention := 'P';
	ELSE SET v_mention := 'R';
	END CASE;
	...

Structures répétitives

Étudions à présent les trois structures répétitives *tant que*, *répéter* et *boucle sans fin*.

Structure tant que

La structure *tant que* se programme à l'aide de la syntaxe suivante. Avant chaque itération (et notamment avant la première), la condition est évaluée. Si elle est vraie, la séquence d'instructions est exécutée, puis la condition est réévaluée pour un éventuel nouveau passage dans la boucle. Ce processus continue jusqu'à ce que la condition soit fausse pour passer en séquence après le `END WHILE`. Quand la condition n'est jamais fausse, on dit que le programme boucle…

```
[etiquette:] WHILE condition DO
    instructions;
END WHILE [etiquette];
```

Le tableau suivant décrit la programmation de deux *tant que*. Le premier calcule la somme des 100 premiers entiers. Le second recherche le premier numéro 4 dans une chaîne de caractères.

Tableau 6-15 Structures tant que

Condition simple	Condition composée
	DECLARE v_telephone CHAR(14) DEFAULT '06-76-85-14-89';

```
DECLARE v_somme INT DEFAULT 0;
DECLARE v_entier SMALLINT DEFAULT 1;

WHILE (v_entier <= 100) DO
 SET v_somme  := v_somme+v_entier;
 SET v_entier := v_entier+1;
END WHILE;
SELECT v_somme;
```

```
+---------+
| v_somme |
+---------+
|    5050 |
+---------+
```

```
DECLARE v_trouve BOOLEAN
        DEFAULT  FALSE;
DECLARE v_indice SMALLINT DEFAULT  1;
WHILE (v_indice <= 14 AND NOT v_trouve) DO
  IF SUBSTR(v_telephone,v_indice,1) = '4'
   THEN
      SET v_trouve := TRUE;
   ELSE
      SET v_indice := v_indice + 1;
  END IF;
END WHILE;
IF v_trouve THEN
   SELECT CONCAT('Trouvé 4 à l''indice :
',v_indice);
END IF;
```

```
Trouvé 4 à l'indice : 11
```

Cette structure est la plus puissante, car elle permet de programmer aussi un *répéter*, une *boucle sans fin* et même un *pour* (qui n'est pas encore opérationnel). Elle doit être utilisée quand il est nécessaire de tester une condition avant d'exécuter les instructions contenues dans la boucle.

Structure répéter

La structure *répéter* se programme à l'aide de la syntaxe `REPEAT… UNTIL`.

Enfin un *répéter* qui se programme comme il faut (à savoir « répéter… jusqu'à *condition* »). Les langages C et Java nous avaient déformé cette traduction par `do {…} while(`*condition*`)` qui nécessite d'écrire l'inverse de la condition du *jusqu'à* de l'algorithmique. Ouf, MySQL (comme Oracle) a bien programmé la structure *répéter* en traduisant ce fameux *jusqu'à* par la directive *until*, et non plus par ce fâcheux *while*.

```
[etiquette:] REPEAT
    instructions;
UNTIL condition END REPEAT [etiquette];
```

La particularité de cette structure est que la première itération est effectuée quelles que soient les conditions initiales. La condition n'est évaluée qu'en fin de boucle.

- Si la condition est fausse, la séquence d'instructions est de nouveau exécutée. Ce processus continue jusqu'à ce que la condition soit vraie pour

passer en séquence après le END REPEAT.

- Quand la condition n'est jamais vraie, on dit aussi que le programme boucle…

Le tableau suivant décrit la programmation de la somme des 100 premiers entiers et de la recherche du premier numéro 4 dans une chaîne de caractères, à l'aide de la structure *répéter*. Les variables sont les mêmes qu'au tableau précédent.

Tableau 6-16 Structures répéter

Condition simple	Condition composée

REPEAT
SET v_somme := v_somme + v_entier;
SET v_entier := v_entier + 1;
UNTIL v_entier > 100 **END REPEAT;**
	REPEAT
	  IF SUBSTR(v_telephone,v_indice,1) = '4'
	  THEN
	  SET v_trouve := TRUE;
	  ELSE
	  SET v_indice := v_indice + 1;
	  END IF;
	UNTIL (v_indice > 14 OR v_trouve)
	  END REPEAT;
	IF v_trouve THEN
	  SELECT CONCAT('Trouvé 4 à l''indice :
	',v_indice);
	END IF;

Cette structure doit être utilisée quand il n'est pas nécessaire de tester la condition avec les données initiales, avant d'exécuter les instructions contenues dans la boucle.

Structure boucle sans fin

La syntaxe générale de cette structure est programmée par la directive LOOP. Elle devient sans fin si vous n'utilisez pas l'instruction LEAVE qui passe en séquence du END LOOP.

```
[etiquette:] LOOP
    instructions;
END LOOP [etiquette];
```

Le tableau suivant donne l'écriture du calcul de la somme des 100 premiers entiers en utilisant deux boucles sans fin (qui se terminent toutefois, car *tout a une fin*, mais celles-là je les programme avec LEAVE). J'en profite pour présenter ITERATE qui force à reprendre l'exécution au début de la boucle.

Tableau 6-17 Structures de boucles sans fin

Avec LEAVE

```
boucle1: LOOP
  SET v_somme  := v_somme + v_entier;
  SET v_entier := v_entier + 1;
  IF v_entier > 100 THEN
     LEAVE boucle1;
  END IF;
END LOOP boucle1;
```

Avec ITERATE

```
boucle1: LOOP
  SET v_somme  := v_somme + v_entier;
  SET v_entier := v_entier + 1;
  IF v_entier <= 100 THEN
      ITERATE boucle1;
  END IF;
  LEAVE boucle1;
END LOOP boucle1;
```

Il est à noter que LEAVE peut être aussi utilisé pour sortir d'un bloc (s'il est étiqueté). LEAVE et ITERATE peuvent aussi être employés au sein de structures REPEAT ou WHILE.

Structure pour

Renommée pour les parcours de vecteurs, tableaux et matrices en tout genre, la structure *pour* se caractérise par la connaissance a priori du nombre d'itérations que le programmeur souhaite faire effectuer à son algorithme. La syntaxe générale de cette structure est programmée dans tous les langages par l'instruction `for`.

La structure de contrôle FOR n'est pas encore implémentée par MySQL. Vous devrez la programmer par un *répéter* (REPEAT…), un *tant que* (WHILE…) ou encore par une *boucle sans fin* (LOOP…). Pour chacune de ces alternatives, vous devrez définir un indice qui évoluera d'une valeur initiale à une valeur finale par une incrémentation en fin de boucle.

Interactions avec la base

Cette section décrit les mécanismes que MySQL offre pour interfacer un sous-programme avec une base de données.

Extraire des données

La principale instruction capable d'extraire des données contenues dans des

tables est SELECT. Étudiée au chapitre 4 dans un contexte SQL, la particularité de cette instruction au niveau d'un sous-programme est la directive INTO qui permet de charger des variables à partir de valeurs de colonnes, comme le montre la syntaxe suivante :

```
SELECT col1 [,col2 ...]INTO variable1 [,variable2 ...]
       FROM nomTable ...;
```

Cette instruction peut aussi être utilisée à l'extérieur d'un bloc pour charger une variable de session, par exemple.

Veillez à ne récupérer qu'un seul enregistrement à l'aide du WHERE de la requête. C'est logique, puisque vous désirez ne charger qu'une valeur par variable.

- Si vous en extrayez plusieurs, vous verrez l'erreur : « ERROR 1172 (42000): Result consisted of more than one row ».
- Si vous n'en extrayez aucun (*no data found*), aucune erreur n'est soulevée et la variable est inchangée (elle reste initialisée à la valeur présente avant la requête).

Colonnes simples

Le tableau suivant décrit l'extraction de la colonne compa pour le pilote de code 'PL-2' dans différents contextes :

Tableau 6-18 Extraction de données

Code MYSQL	Commentaires

BEGIN
  DECLARE v_comp VARCHAR(15);
  SELECT compa **INTO** v_comp
  FROM Pilote WHERE brevet='PL-2';
  ...
END;
	Chargement d'une variable locale à un bloc. Nécessité d'appeler par la suite cette procédure (CALL).

SET @vs_compa=''$ SELECT compa **INTO** @vs_compa FROM Pilote WHERE brevet='PL-2'$ ...	Chargement d'une variable de session hors d'un sous-programme.

SET @vs_compa=''$
CREATE PROCEDURE sp1()
BEGIN
  SELECT compa **INTO** @vs_compa
  FROM Pilote WHERE brevet='PL-2';
	Chargement d'une variable de session dans un sous-programme.

```
   ...
END;
```

Pour traiter des requêtes renvoyant plusieurs enregistrements, il faudra utiliser des curseurs (étudiés au chapitre suivant).

Fonctions SQL

Il est naturel que les fonctions SQL (mono et multilignes) étudiées au chapitre 4 soient également disponibles dans un sous-programme, à condition de les utiliser au sein d'une instruction SELECT. Deux exemples sont décrits dans le tableau suivant : le premier chargera la variable avec le nom du pilote de code 'PL-1' en majuscules (table décrite au début du chapitre 4) ; le second affectera à la variable le maximum du nombre d'heures de vol, tous pilotes confondus.

Tableau 6-19 Utilisation de fonctions

Monoligne	Multiligne

`BEGIN`
`  DECLARE v_nomEnMAJUSCULES CHAR(20);`
`  SELECT UPPER(nom)`
`  INTO v_nomEnMAJUSCULES`
`  FROM Pilote WHERE brevet = 'PL-1';`
`SELECT v_nomEnMAJUSCULES;`
`END;`
	`BEGIN`
	`  DECLARE v_plusGrandHVol DECIMAL(7,2);`
	`  SELECT MAX(nbHVol) INTO v_plusGrandHVol`
	`  FROM Pilote;`
	`SELECT v_plusGrandHVol ;`
	`END;`

`+------------------+`
`| v_nomEnMAJUSCULES |`
`+------------------+`
`| GRATIEN VIEL  |`
`+------------------+`
	`+-----------------+`
	`| v_plusGrandHVol |`
	`+-----------------+`
	`|  2450.00 |`
	`+-----------------+`

Manipuler des données

Les principales instructions disponibles pour manipuler, par un sous-programme, les éléments d'une base de données sont les mêmes que celles proposées par SQL, à savoir INSERT, UPDATE et DELETE. Pour libérer les verrous au niveau d'un enregistrement (et des tables), il faudra ajouter les instructions COMMIT ou ROLLBACK (aspects étudiés en fin de chapitre).

Insertions

Le tableau suivant décrit l'insertion de différents enregistrements sous plusieurs écritures (il est aussi possible d'utiliser des variables de session) :

Tableau 6-20 Insertion d'enregistrements

Code MySQL	Commentaires

`BEGIN`
` DECLARE v_brevet VARCHAR(6) DEFAULT 'PL-7';`
` DECLARE v_nom VARCHAR(6);`
` DECLARE v_HVol DECIMAL(7,2) DEFAULT 0;`
` DECLARE v_comp VARCHAR(6);`
	Déclaration des variables locales au bloc.

INSERT INTO Pilote VALUES
`  ('PL-6', 'Jules Ente', 3000, 'AF');`
` SET v_nom := 'Fabrice Peyrard';`
` SET v_comp := 'SING';`
	Insertion d'un enregistrement en renseignant les colonnes par des constantes.

INSERT INTO Pilote VALUES
`  (v_brevet,v_nom,v_HVol,v_comp);`
`END;`
	Insertion d'un enregistrement en renseignant les colonnes par des variables locales.

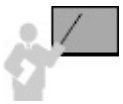
Comme sous SQL, il faut respecter les noms, types et domaines de valeurs des colonnes. De même, les contraintes de vérification (CHECK qui n'est pas encore opérationel et NOT NULL) et d'intégrité (PRIMARY KEY et FOREIGN KEY) doivent être valides.

Dans le cas inverse, une exception qui précise la nature du problème est levée et peut être interceptée par la directive HANDLER (voir chapitre suivant). Si une telle directive n'existe pas dans le bloc qui contient l'instruction INSERT, la première exception fera s'interrompre le programme.

Modifications

Concernant la mise à jour de colonnes par UPDATE, la clause SET peut être ambiguë dans le sens où l'identificateur à gauche de l'opérateur d'affectation est toujours une colonne de base de données, alors que celui à droite de l'opérateur peut correspondre à une colonne ou à une variable.

```
UPDATE nomTable
  SET col1 = { variable1 | expression1 | autrecol | (requête) }
    [,col2 = ...]
  [WHERE ... ];
```



Si aucun enregistrement n'est modifié, aucune erreur ne se produit et aucune

exception n'est levée.

Alors que les affectations dans le code MYSQL (SET …) peuvent s'écrire par les symboles « := » ou « = », les comparaisons ou affectations SQL nécessitent le symbole « = ».

Le tableau suivant décrit la modification de différents enregistrements (il est aussi possible d'employer des variables de session).

Tableau 6-21 Modifications d'enregistrements

Code MySQL	Commentaires
`BEGIN` `DECLARE v_dureeVol DECIMAL(3,1) DEFAULT 4.8;`	Déclaration.

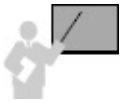
`UPDATE Pilote`
`SET nbHVol= nbHVol + v_dureeVol`
`WHERE brevet= 'PL-6';`
	Modification d'un enregistrement de la table `Pilote` en utilisant une variable.

`UPDATE Pilote`
`SET nbHVol= nbHVol + 10`
`WHERE compa = 'AF';`
`END;`
	Modification de plusieurs enregistrements de la table `Pilote` en utilisant une constante.

Suppressions

La suppression par DELETE peut être ambiguë (même raison que pour l'instruction UPDATE) au niveau de la clause WHERE.

```
DELETE FROM nomTable
  [WHERE col1 = { variable1 | expression1 | autrecol | (requête) }
    [,col2 = ...] ];
```



Si aucun enregistrement n'est modifié, aucune erreur ne se produit et aucune exception n'est levée.

Le tableau suivant décrit la suppression de différents enregistrements (il est aussi possible d'utiliser des variables de session).

Tableau 6-22 Suppression d'enregistrements

Code MYSQL	Commentaires

`BEGIN`
`DECLARE v_hVolMini DECIMAL(7,2) DEFAULT 1000.00;`
`DELETE FROM Pilote WHERE nbHVol < v_hVolMini;`	

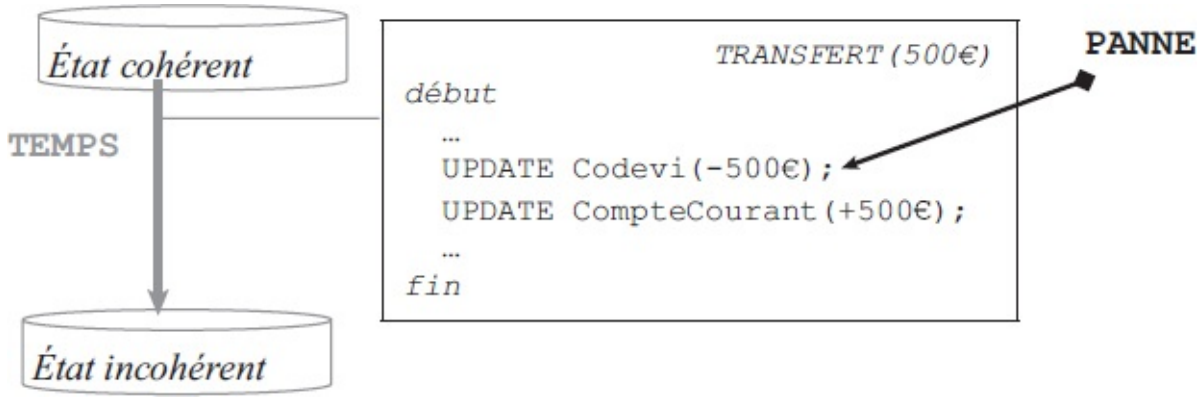
`DELETE FROM Pilote WHERE brevet = 'PL-3';`	Supprime les enregistrements de la table `Pilote` dont le nombre d'heures de vol est inférieur à 1 000. Supprime un pilote.
`DELETE FROM Pilote WHERE brevet = NULL;` `END;`	Ne supprime aucun pilote.

Gestion des transactions

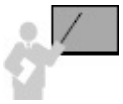
Au sens SGBD du terme, une transaction est un bloc d'instructions LMD faisant passer la base de données d'un état initial (cohérent) à un état intermédiaire ou final cohérent. Si un problème logiciel ou matériel survient au cours d'une transaction, aucune des instructions de la transaction ne doit être effectuée. En invalidant toutes les opérations depuis le début de la transaction, la base retourne à son état initial cohérent (suivant le principe du tout ou rien).

Un exemple typique de transaction consiste au transfert d'une somme d'un compte épargne vers un compte courant. Imaginez qu'après une panne votre compte épargne ait été débité sans que votre compte courant soit crédité du même montant ! Vous ne seriez pas très content des services de votre banque (à moins que l'erreur soit survenue dans l'autre sens). La réservation d'une place au théâtre ne permet pas non plus que plusieurs personnes partagent le même siège…

Figure 6-5 *Procédure fâcheuse*



Le mécanisme transactionnel empêche tout scénario problématique par la technique de journalisation et celle des verrous.

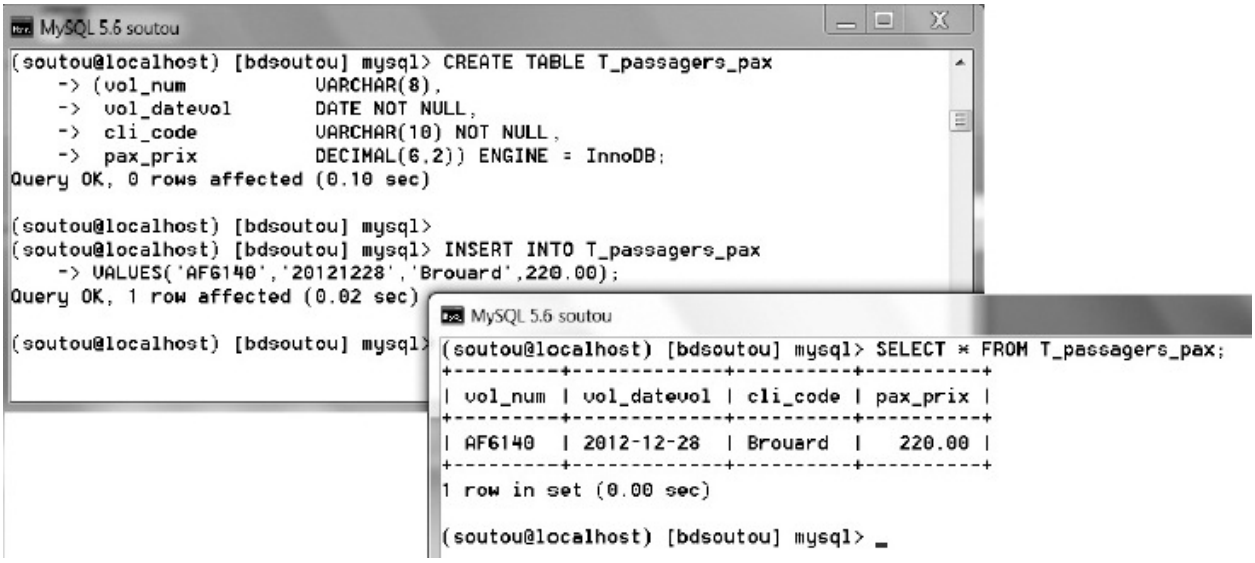


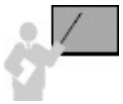
Dans un mode transactionnel, une extraction (SELECT) génère un verrou partagé (S) sur tout ou partie de la table. Si une écriture (UPDATE ou MERGE) concerne cette partie de table, un verrou exlusif (X) est posé et il sera impossible d'obtenir le verrou (S) tant que la modification n'est pas validée. S'il s'agissait d'une autre lecture, la requête initiale pourra s'exécuter. L'idée de base est qu'une lecture ne doit pas bloquer une autre lecture, mais qu'une écriture peut bloquer une autre écriture (ou lecture) et qu'une lecture peut bloquer une écriture.

Début et fin d'une transaction

Dans l'exemple suivant, la première session (qui compose une transaction courante) crée une table et insère une ligne sans validation explicite. La deuxième session (qui compose une seconde transaction courante) accède aux données.

Figure 6-6 *Transaction implicite*

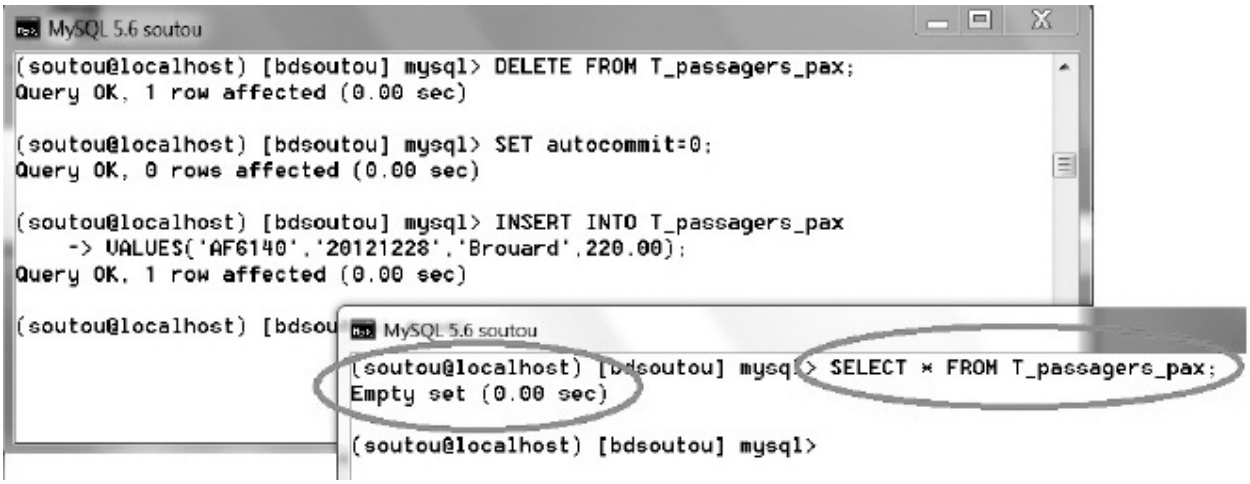




La norme prévoit que toute connexion à un SGBD crée une nouvelle transaction. C'est faux avec MySQL qui travaille par défaut en mode `autocommit`, c'est-à-dire que chaque instruction SQL du LMD qui se termine avec succès est automatiquement validée.

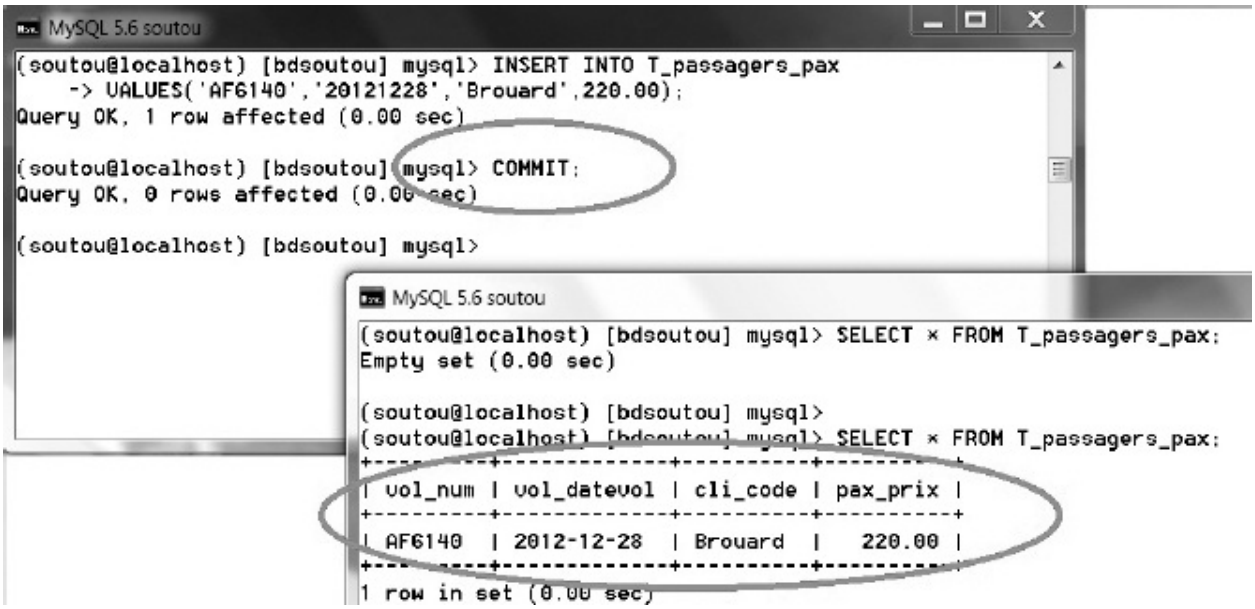
En annulant l'autovalidation implicite dans la première transaction (`SET autocommit=0`), on insère de nouveau une ligne sans validation explicite. La deuxième transaction ne peut accéder aux données (qui sont toutefois visibles par la première transaction).

***Figure 6-7** Transaction non validée*



Dès que la première session valide explicitement la transaction, l'accès aux données est possible par toute nouvelle transaction qui démarre après – il est entendu ici que chaque instruction constitue une transaction car nous n'avons pas encore abordé les transactions au sens large, à savoir celles qui sont composées d'un bloc d'instructions.

Figure 6-8 *Transaction validée*

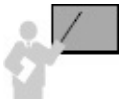


Pour maîtriser votre code et vous prémunir des incohérences dues à des accès concurrents, vous pouvez programmer vos transactions explicitement à l'aide des primitives décrites au tableau suivant.

Tableau 6-23 Principales instructions des transactions

Instruction	Objectif
`SET AUTOCOMMIT = {0 | 1}`	Change le mode de validation (1 par défaut, 0 à adopter pour contrôler une transaction)
`SET TRANSACTION ISOLATION LEVEL [ option ];`	Mise en place du niveau d'isolation (étudié plus loin)
`START TRANSACTION [ options ];`	Début de la transaction (avec options possibles qui seront étudiées plus loin)

`SAVEPOINT [ nom_savepoint ];`	Déclare un point de validation au cours de la transaction
`COMMIT [ options ];`	Termine avec succès la transaction (validation)
`ROLLBACK [ options ];`	Termine avec échec la transaction (invalidation)



Vous devrez employer la commande `START TRANSACTION` (ou `BEGIN`) qui démarre une transaction explicite et désactive le mode `autocommit` jusqu'à la fin de la transaction, soit par `COMMIT` (si le déroulement du code est satisfaisant), soit par `ROLLBACK` (pour défaire toutes les modifications opérées depuis le début). Ensuite, le naturel revient au galop car le mode `auto-commit` revient à son état d'origine (par défaut en mode autovalidation à 1 mais reste à 0 s'il y avait été positionné avant).

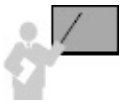
Il n'est pas possible d'invalider par `ROLLBACK` une commande LDD ou LCD (`CREATE`, `ALTER`, `DROP`...).

Le tableau suivant illustre une transaction validée qui insère un passage, et une autre qui suit, non validée du fait du positionnement de la variable `autocommit`.

Tableau 6-24 Début et fin d'une transaction validée

Session 1	Session 2 (qui s'exécute après la session 1)

`DELETE FROM T_passagers_pax;`
`SET autocommit=0;`
`START TRANSACTION``;`
`INSERT INTO T_passagers_pax VALUES`
`('AF6140','20121228','Brouard',220.00);`
`COMMIT``;`
`INSERT INTO T_passagers_pax VALUES`
`('AF6140','20121228','Roux',189.00);`
	`mysql> SELECT * FROM T_passagers_pax;`
	`+-------+-----------+--------+--------+`
	`|vol_num|vol_datevol|cli_code|pax_prix|`
	`+-------+-----------+--------+--------+`
	`|AF6140 |2012-12-28 |Brouard |  220.00|`
	`+-------+-----------+--------+--------+`



Si une nouvelle instruction `START TRANSACTION` (ou `BEGIN`) arrive au milieu du code d'une transaction explicite, la transaction initiale est validée et une nouvelle démarre (il n'y a pas d'imbrication de transactions).

Toute transaction se termine implicitement :

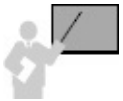
- avec succès à la première commande SQL du LDD rencontrée (`CREATE`, `ALTER`…) ;
- avec insuccès à la fin normale (`quit` ou `exit`) ou anormale d'une session (arrêt forcé du processus).

Gestion des anomalies transactionnelles

La concurrence d'accès aux données induit des problèmes inévitables. Prises isolément et exécutées les unes après les autres, des transactions modifiant des données en commun ne généreront aucune incohérence. Le problème est que ces transactions sont susceptibles de s'exécuter en même temps. Sans parler de parallélisme, les traitements multitâches permettent l'entrelacement non contrôlé des opérations. La mise en place d'un niveau d'isolation pour chaque transaction permet de gérer au mieux cet état de fait.

Chaque niveau sert à résoudre un type d'anomalie et est implémenté par deux mécanismes : pose de différents verrous et exploitation de la journalisation des opérations réalisées par chaque transaction. Trois types d'anomalies sont recensés :

- la lecture de données sales *(dirty reads)* qui se produit lorsqu'une transaction accède à des données qui sont manipulées par une autre transaction qui n'a pas été encore validée ;
- la lecture non renouvelable de données *(non repeatable reads)* qui se produit quand deux lectures successives d'une même donnée au sein d'une transaction ne produisent pas le même résultat (donnée manipulée par une autre transaction) ;
- la lecture de données fantômes *(phantom reads)* qui se produit lorsque de nouvelles données apparaissent ou disparaissent au cours de lectures successives (données manipulées par une autre transaction).



Depuis MySQL 5.6.5, la mise en place du niveau de transaction (qui n'est valable que pour les tables InnoDB) se fait par la commande `SET TRANSACTION ISOLATION LEVEL` qui doit se trouver avant le début de la transaction (`START TRANSACTION` ou `BEGIN`). L'option `GLOBAL` caractérisera toutes les futures transactions, qu'elles appartiennent à la session ou pas. L'option `SESSION` (par défaut) concernera les transactions de la session en cours.

Les modes disponibles sont :

```
SET [GLOBAL | SESSION] TRANSACTION ISOLATION LEVEL
 {READ UNCOMMITTED | READ COMMITTED | REPEATABLE READ | SERIALIZABLE};
```

Le comportement par défaut d'une transaction est `REPEATABLE READ`.

Selon le mode choisi, vous disposerez, en théorie (selon la norme SQL), des caractéristiques suivantes.

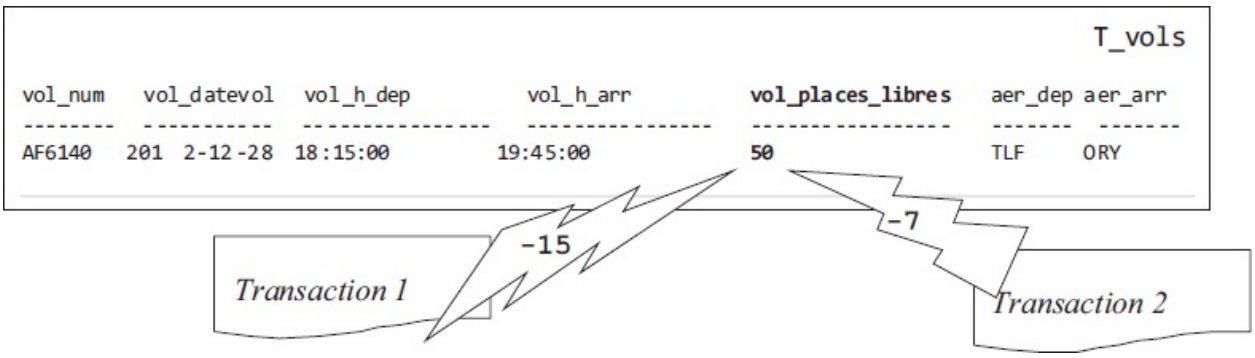
Tableau 6-25 Niveaux d'isolation (norme SQL)

Niveau d'isolation	Lectures sales	Lectures non renouvelables	Lectures fantômes
`READ UNCOMMITTED`	Possible	Possible	Possible
`READ COMMITTED`	Impossible	Possible	Possible
`REPEATABLE READ`	Impossible	Impossible	Possible
`SERIALIZABLE`	Impossible	Impossible	Impossible

À titre de comparaison, Oracle, qui est considéré comme le plus performant en termes de gestion transactionnelle, ne fournit que les niveaux `READ COMMITTED` et `SERIALIZABLE` qui sont les plus couramment employés.

L'exemple suivant correspond à la réservation de places pour un vol entre Toulouse et Paris. Supposons qu'il reste 50 places disponibles et que deux transactions tentent de réserver simultanément 7 places pour l'une et 15 places pour l'autre. L'état idéal serait que ces deux transactions laissent au final 28 places disponibles pour le vol…

Figure 6-9 *Accès concurrents par deux transactions en action*



Dans les scripts qui suivent, l'instruction `SELECT SLEEP(…)` permet de simuler une concurrence d'accès en mettant en attente un traitement.

Le mode `READ UNCOMMITTED`

Les transactions qui utilisent ce mode ne mettent en œuvre aucun verrou partagé (ni exclusif) pour empêcher d'autres transactions de modifier des données exploitées entre elles.

Le tableau suivant présente une lecture sale en simulant une transaction qui réserve 7 places sans valider finalement, et la seconde en réserve 15 en validant. Le problème vient du fait d'absence de verrous qui fait lire à la transaction 2 qu'il reste 43 places. On suppose ici que la transaction 1 démarre avant la transaction 2. Au final, 28 places sont disponibles, alors que seules 15 places ont été réservées, un sacré manque à gagner…

Tableau 6-26 Exemple de transactions sans verrouillage

Transaction 1	Transaction 2
START TRANSACTION; DECLARE v_reste TINYINT; DECLARE v_resa TINYINT; SET v_resa := 7; SELECT vol_places_libres INTO v_reste FROM T_vols WHERE vol_num = 'AF6140' AND vol_datevol ='2012-12-28';	START TRANSACTION; DECLARE v_reste TINYINT; DECLARE v_resa TINYINT; SET v_resa := 15; SELECT vol_places_libres INTO v_reste FROM T_vols WHERE vol_num = 'AF6140' AND vol_datevol ='2012-12-28';
UPDATE T_vols SET vol_places_libres = v_reste - v_ resa WHERE vol_num = 'AF6140' AND vol_datevol ='2012-12-28';	
SELECT SLEEP(10);	UPDATE T_vols SET vol_places_libres = v_reste - v_resa WHERE vol_num = 'AF6140' AND vol_datevol ='2012-12-28';
ROLLBACK;	COMMIT;

```
SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;

vol_places_libres
-----------------
               28
```

Pour éviter ce fâcheux résultat, il suffit de définir ces transactions à un niveau supérieur (le premier étant `READ COMMITTED`). Des verrous seront posés pour empêcher toute modification de lignes non validées.

Le mode `READ COMMITTED`

Dans le mode `READ COMMITTED`, les transactions n'ont pas accès en lecture aux données modifiées mais non validées. Toutefois, les données partagées peuvent être modifiées par d'autres transactions pour aboutir à des lectures non renouvelables ou fantômes.

Le tableau suivant présente une lecture non renouvelable en simulant la transaction 1 qui réserve 7 places, et la seconde qui interroge successivement la même colonne en retournant deux résultats distincts (50 puis 43). On suppose que la seconde transaction démarre avant la première. Dans ce cas précis, l'anomalie peut ne pas être considérée en tant que telle mais l'isolation n'est pas vraiment garantie (le mode `SERIALIZABLE` évitera ce comportement).

Tableau 6-27 Transactions en mode `read committed`

Transaction 1	Transaction 2

`START TRANSACTION;`
` DECLARE v_reste TINYINT;`
` DECLARE v_resa  TINYINT;`
` SET     v_resa  := 7;`
` UPDATE T_vols`
`   SET vol_places_libres =`
`      vol_places_libres - v_resa`
`    WHERE vol_num       = 'AF6140'`
`    AND   vol_datevol   = '2012-12-28';`
`COMMIT;`
	`START TRANSACTION;`
	` DECLARE v_reste TINYINT;`
	` SELECT vol_places_libres INTO v_reste`
	`    FROM T_vols`
	`    WHERE vol_num     = 'AF6140'`
	`    AND   vol_datevol = '2012-12-28';`
	` SELECT CONCAT('disponible : ',v_reste);`
	` SELECT SLEEP(10);`
	` SELECT vol_places_libres INTO v_reste`
	`    FROM T_vols`
	`    WHERE vol_num     = 'AF6140'`
	`    AND   vol_datevol = '2012-12-28';`
	` SELECT CONCAT('disponible : ',v_reste);`
	`ROLLBACK;`

```
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;

disponible : 50
disponible : 43
```

Le mode `REPETEABLE READ`

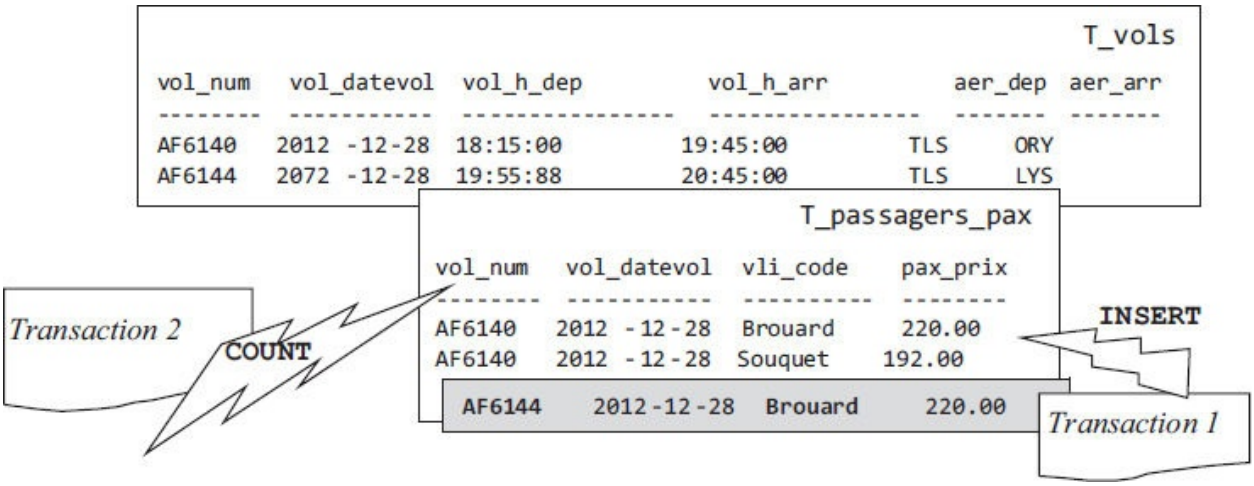
En configurant la session 2 avec SET TRANSACTION ISOLATION LEVEL REPEATABLE READ, vous éviterez que deux lectures successives de la même donnée renvoient deux résultats différents. Pour jouer correctement ce scénario, démarrez la transaction 2 avant la transaction 1. Bien que la réservation ait été validée par la transaction 1, la vue des données du côté de la transaction 2 sera inchangée (50 places), l'isolation est davantage garantie.

En théorie, le mode REPEATABLE READ n'évite pas, pendant la transaction, que de nouvelles données apparaissent ou d'anciennes disparaissent du fait de transactions concurrentes (données fantômes).

La lecture fantôme

La figure suivante présente la transaction 2 qui compte le nombre de vols de chaque passager. Dans l'intervalle, la transaction 1 ajoute un vol à un passager. Quand la transaction 2 débute avant la transaction 1 et se prolonge dans le temps, de nouveaux vols peuvent apparaître au cours du traitement, ce qui ne reflète pas l'état de la base au début de la transaction de lecture.

Figure 6-10 *Lecture fantôme entre deux transactions en action*



Deux alternatives s'offrent à vous :

- vous souhaitez accéder le plus rapidement aux dernières mises à jour et cela ne risque pas d'altérer votre traitement : restez en mode READ COMMITED ou REPEATABLE READ ;
- vous souhaitez que l'isolation entre les transactions soit totale et que votre traitement ne soit pas pollué par d'autres transactions même si elles sont

validées dans l'intervalle : passez en mode SERIALIZABLE.

La lecture consistante

Le mode SERIALIZABLE assure un mécanisme de lecture consistante *(read consistency)* masquant les mises à jour concurrentes et donc préservant l'état initial de la base avant chaque transaction. Avec ce niveau d'isolation, toute transaction ne peut avoir accès qu'à des données qui ont été validées avant son début. Les mises à jour concurrentes sont toutefois possibles mais transparentes aux autres transactions.

Le mode SERIALIZABLE pose des verrous et ne les relâche qu'à la fin de la transaction. On dit que ce mode repose sur un contrôle pessimiste de la concurrence (deux transactions peuvent modifier les données au même moment au risque qu'une transaction doive attendre indéfiniment l'autre : *deadlock*). Un ensemble de transactions de mode SERIALIZABLE s'exécute comme s'il s'agissait de transactions entrelacées ou par un ordre séquentiel, quel qu'il soit.

Le tableau suivant présente les deux transactions dont la deuxième est isolée de la première et ne voit pas l'ajout d'un vol à un passager (évite une lecture fantôme).

Tableau 6-28 Lecture consistante

Transaction 1	Transaction 2
START TRANSACTION; INSERT INTO T_passagers_pax VALUES ('AF6144','20121228','Brouard',220.00); COMMIT;	START TRANSACTION; DECLARE v_nb_vol TINYINT; SELECT COUNT(vol_num) INTO v_nb_vol FROM T_passagers_pax WHERE cli_code='Brouard'; SELECT CONCAT('nb : ', v_nb_vol); SELECT SLEEP(15); SELECT COUNT(vol_num) INTO v_nb_vol FROM T_passagers_pax WHERE cli_code='Brouard'; SELECT CONCAT('nb : ', v_nb_vol); COMMIT;
	SET TRANSACTION ISOLATION LEVEL SERIALIZABLE; nb : 1 nb : 1

Après moult tests et consultations sur le Net, il semblerait que le mode `REPEATABLE READ` de MySQL interdise les lectures fantômes (le mode `SERIALIZABLE` permet d'éviter ce comportement).

Transactions en lecture seule

Depuis MySQL 5.6.5, les options `READ ONLY` et `READ WRITE`, et la commande `SET TRANSACTION ISOLATION LEVEL` permettent de modifier le mode d'accès des transactions concurrentes. Ceci n'est valable que pour les tables InnoDB. Une transaction déclarée `READ ONLY` interdira les mises à jour par d'autres transactions, des tables qui sont manipulées par la transaction principale. Le mode `READ WRITE` rétablit le comportement par défaut.

Tableau 6-29 Instructions pour les transactions en lecture seule

Instruction	Action
`START TRANSACTION { READ ONLY | READ WRITE };`	Débute une transaction en lecture seule ou en lecture écriture.
`SET TRANSACTION ISOLATION LEVEL` `{ READ ONLY | READ WRITE };`	Modifie le comportement d'une transaction en lecture seule ou en lecture écriture.

Le problème du verrou mortel [*deadlock*]

Le phénomène de *deadlock*, aussi appelé « étreinte fatale » se produit lorsque deux (ou plus) transactions ont posé des verrous sur des objets distincts, et que chacune tente d'acquérir un nouveau verrou sur un objet que l'autre transaction a déjà verrouillé. En plus du fait que les verrous mortels nécessitent d'être gérés comme des exceptions, ils sont gourmands en ressources CPU.

Le tableau suivant illustre deux transactions en interblocage (et ce quel que soit le niveau d'isolation basé sur des verrous). En supposant que la transaction 1 (*t1*) démarre avant la transaction 2 (*t2*), le vol AF6140 est d'abord verrouillé

par *t1* jusqu'à la validation, ce qui n'empêche pas le verrouillage du vol AF6144 par *t2*. Par la suite, *t1* pose un verrou sur le vol AF6144 qui sera relâché à la fin de *t2* qui a posé un verrou sur le vol AF6140 qui lui sera relâché à la fin de *t1*. Tout le monde attend ainsi l'autre.

Tableau 6-30 Transactions en interblocage

Transaction 1	Transaction 2

`START TRANSACTION;`
` UPDATE T_vols`
`  SET vol_places_libres =`
`  vol_places_libres - 7`
`  WHERE vol_num = 'AF6140'`
`  AND vol_datevol ='2012-12-28';`
` SELECT SLEEP(5);`
` UPDATE T_vols`
`  SET vol_places_libres =`
`  vol_places_libres - 4`
`  WHERE vol_num = 'AF6144'`
`  AND vol_datevol ='2012-12-28';`
`COMMIT;`
	`START TRANSACTION;`
	` UPDATE T_vols`
	`  SET vol_places_libres =`
	`  vol_places_libres - 5`
	`  WHERE vol_num = 'AF6144'`
	`  AND vol_datevol ='2012-12-28';`
	` SELECT SLEEP(5);`
	` UPDATE T_vols`
	`  SET vol_places_libres =`
	`  vol_places_libres - 3`
	`  WHERE vol_num = 'AF6140'`
	`  AND vol_datevol ='2012-12-28';`
	`COMMIT;`

Quand MySQL identifie un tel phénomène, il met fin à une des deux transactions. Le message `ERROR 1213 (40001): Deadlock found when trying to get lock; try restarting transaction` est retourné au client malheureux tandis que l'autre transaction termine son exécution.

Adoptez les règles suivantes pour limiter les risques de verrous mortels :

- normalisez le plus possible votre modèle de données (voir l'ouvrage intitulé *UML2 pour les bases de données*) ;
- indexez correctement vos tables (voir le chapitre 11) ;
- accédez à vos objets (tables ou vues) toujours dans le même ordre, par exemple selon l'ordre alphabétique ;
- réduisez la durée du code situé dans un bloc (ni saisies ni entrées-sorties) ;
- si cela est approprié, utilisez un niveau d'isolation plus faible ou gérez manuellement les verrous.

Verrouillage manuel

La gestion manuelle des verrous évite les étreintes fatales mais est complexe à mettre en œuvre et à maintenir et peut se révéler très pénalisante (les transactions concurrentes peuvent être mises systématiquement en attente). Le moyen de procéder est le suivant :

- déclarez la session sans validation automatique (`SET autocommit=0`) et n'utilisez pas `START TRANSACTION` (ou `BEGIN`) ;
- utilisez la commande `LOCK TABLES {READ | WRITE}` sur une ou plusieurs tables InnoDB ;
- après avoir validé (par `COMMIT`) ou invalidé (par `ROLLBACK`), n'oubliez pas d'utiliser la commande `UNLOCK TABLES` pour relâcher les verrous posés.

À titre d'exemple, l'interblocage présenté précédemment serait résolu en écrivant la transaction 1 en suivant ce principe.

Tableau 6-31 Transaction avec verrouillage manuel

Transaction 1	Transaction 2

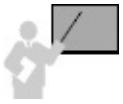
SET autocommit=0;
LOCK TABLES T_vols WRITE;
  UPDATE T_vols
  SET vol_places_libres =
  vol_places_libres - 7
  WHERE vol_num = 'AF6140'
  AND vol_datevol ='2012-12-28';
SELECT SLEEP(5);
UPDATE T_vols
  SET vol_places_libres =
  vol_places_libres - 4
  WHERE vol_num = 'AF6144'
  AND vol_datevol ='2012-12-28';
COMMIT;
UNLOCK TABLES;
	SET autocommit=0;
	UPDATE T_vols
	  SET vol_places_libres =
	  vol_places_libres - 5
	  WHERE vol_num = 'AF6144'
	  AND vol_datevol ='2012-12-28';
	SELECT SLEEP(5);
	UPDATE T_vols
	  SET vol_places_libres =
	  vol_places_libres - 3
	  WHERE vol_num = 'AF6140'
	  AND vol_datevol ='2012-12-28';
	COMMIT;

Ne comptez pas procéder ainsi pour écrire vos procédures cataloguées, car le verrouillage explicite n'est pas permis: `ERROR 1314 (0A000): LOCK is not allowed in stored procedures`.

MySQL propose le paramètre `innodb_lock_wait_timeout` qui permet de limiter l'attente d'une ressource, pendant un délai déterminé, pour une session courante. La valeur par défaut est de 50 secondes.

Contrôle des transactions

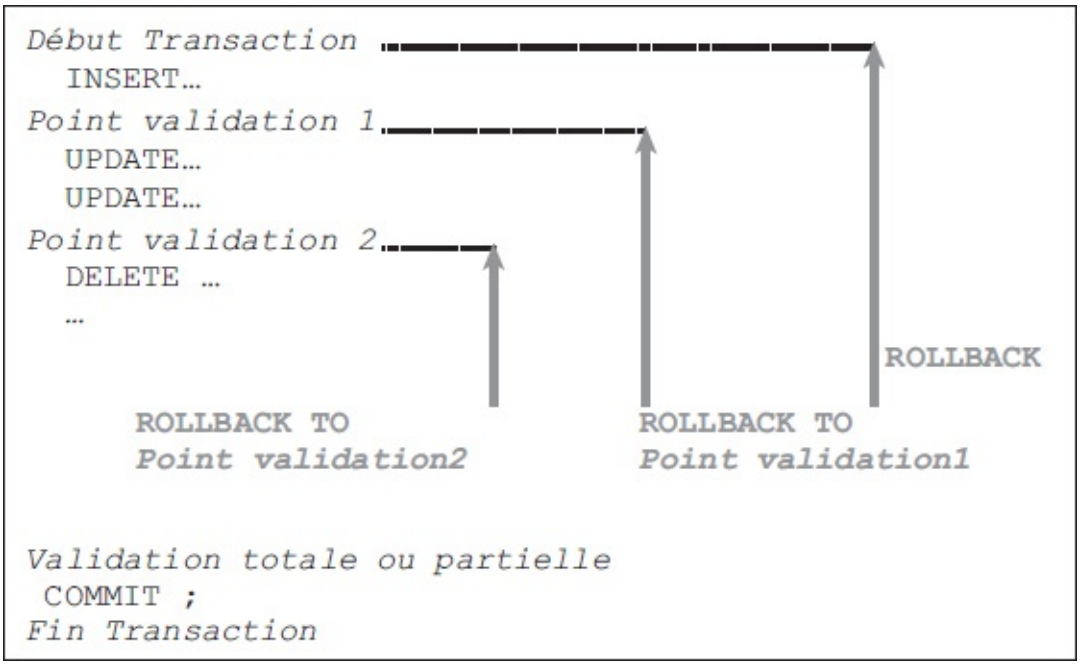
Il peut être intéressant de pouvoir découper une transaction en insérant des points de validation *(savepoints)* qui rendent possible l'annulation de tout ou partie des opérations composant ladite transaction.



L'instruction SAVEPOINT déclare un point de validation.

La figure suivante illustre une transaction découpée en trois parties. Les différentes instructions ROLLBACK permettent d'invalider tout ou partie de la transaction, en laissant la possibilité de valider des sous-parties par un éventuel COMMIT final.

Figure 6-11 *Points de validation*



Le tableau suivant décrit une transaction MySQL découpée en plusieurs parties. Les points de validation et les annulations jusqu'à différents points permettent de valider tout ou partie de la transaction.

Tableau 6-32 Transaction découpée

Code MYSQL	Commentaires

`START TRANSACTION`
`INSERT INTO Compagnie VALUES`
`('C2',2, 'Place Brassens', 'Blagnac', 'Easy Jet');`
	Première partie de la transaction.

`SAVEPOINT P1;`
`UPDATE Compagnie SET nrue = 125`
`WHERE comp = 'AF';`
`UPDATE Compagnie SET ville = 'Castanet'`
`WHERE comp = 'C1';`
	Deuxième partie de la transaction.

`SAVEPOINT P2;` `DELETE FROM Compagnie WHERE comp = 'C1';`	Troisième partie de la transaction.
`SAVEPOINT P3;`	Point de validation.
`ROLLBACK TO SAVEPOINT P1;`	Première partie à valider.
`ROLLBACK TO SAVEPOINT P2;`	Deuxième partie à valider.
`ROLLBACK TO SAVEPOINT P3;`	Troisième partie à valider.
`/* autres mises à jour */`	
`/* ROLLBACK; */`	Tout à invalider.
`COMMIT;`	Valide la ou les sous-parties.

Il n’est pas possible d’imbriquer plusieurs transactions se déroulant dans différents blocs.

Quel mode adopter ?

Bien que l’intégrité des données soit primordiale, il n’est pas souvent utile de rechercher un niveau d’isolation supérieur à celui fourni par défaut (REPEATABLE READ) et qui convient à la majorité des traitements. Plus le niveau d’isolation choisi est fort, plus il risque d’entraîner des contentions relatives au verrouillage. À titre d’information, les applications NoSQL adoptent un mode READ UNCOMMITTED misant sur la très faible probabilité d’une mise à jour simultanée des données.

Pour finir, sachez que le mode READ COMMITTED est moins utilisé que REPEATABLE-READ (de fait que ce dernier est le mode par défaut), il est également moins testé et donc peut être un peu moins fiable. Des experts s’accordent à dire que la plupart des verrous mortels se retrouvent dans les transactions en mode READ

COMMITTED.

Où placer les transactions ?

L'idée de manipuler des transactions depuis un code client (VB, Delphi, Java, C++…) est séduisante mais peut entraîner un blocage du serveur du fait d'une non-libération des verrous (si le client perd la connexion sans validation ou invalidation). Un autre problème concerne les entrées-sorties qui peuvent devenir également blocantes.

La logique transactionnelle doit donc se trouver au plus près du serveur. Et qui mieux que les procédures stockées peuvent implémenter les transactions ? Il est aussi possible d'utiliser des objets métier dédiés (EJB par exemple dans une architecture J2EE).

Modes d'exécution SQL

Le mode SQL permet de régir, dans une certaine mesure, le comportement du serveur face à telle ou telle instruction SQL. Ce comportement correspond principalement à la syntaxe et aux vérifications que doit assurer (ou pas) le serveur.

Cette fonctionnalité peut être intéressante pour tester le contexte de production sur des environnements différents ou avec des bases de données hétérogènes. Préparer une migration vers un autre SGBD peut aussi être considéré comme un objectif des modes SQL. Un autre domaine d'application concerne le traitement de données en masse où il peut être utile de désactiver des contrôles basiques pour améliorer les performances.

Le contexte

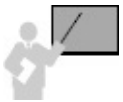
Le mode peut se définir pour toutes les sessions au démarrage du serveur par `mysqld --sql-mode="mode1,mode2…"`. Vous pouvez également agir au niveau du fichier d'initialisation (`my.cnf` sous Unix et `my.ini` sous Windows) à l'aide de l'étiquette `sql-mode="mode1,mode2…"`. Le mode par défaut est le mode vide.

Au niveau d'une ou de plusieurs sessions, il est possible de définir le mode d'exécution courant à l'aide de la commande `SET [GLOBAL|SESSION]`

`sql_mode='mode1,mode2…'`. Vous devrez détenir le privilège `SUPER` pour bénéficier de l'option `GLOBAL` (qui affectera toutes les sessions qui démarreront après). Dans une session, pour connaître le mode SQL courant de la session et le mode plus global, il suffit d'interroger cette variable système.

```
mysql> SELECT @@GLOBAL.sql_mode "Global", @@SESSION.sql_mode "Session";
+-------------------------------------------------------------------------------+
| Global                          | Session                                     |
+-------------------------------------------------------------------------------+
| STRICT_TRANS_TABLES,            | STRICT_TRANS_TABLES,                        |
| NO_AUTO_CREATE_USER,            | NO_AUTO_CREATE_USER,                        |
| NO_ENGINE_SUBSTITUTION          | NO_ENGINE_SUBSTITUTION                      |
+-------------------------------------------------------------------------------+
```

Présentons quelques-unes des options disponibles.



L'expression « mode SQL strict » s'applique aux modes qui sont `TRADITIONAL`, `STRICT_TRANS_TABLES`, ou `STRICT_ALL_TABLES`. Ce mode est à préconiser surtout pour les transactions.

Programmation transactionnelle

Les modes `STRICT_TRANS_TABLES` (par défaut depuis la version 5.7) et `STRICT_ALL_TABLES` déclenchent l'invalidation de l'instruction dès lors qu'une mise à jour d'une table transactionnelle (moteur InnoDB) pose problème (mauvais type de données, mauvaise valeur, valeur manquante ou nulle, etc.). Pour les tables non transactionnelles, l'instruction est invalidée si une seule ligne est concernée ou si c'est la première ligne modifiée d'une instruction multiple (*multiple-row statement*).

- Lorsque l'erreur survient dès la deuxième ligne, le comportement `STRICT_ALL_TABLES` retournera une erreur et ignorera le reste des modifications (tout en validant les premières). Pour éviter ceci, essayez de ne pas utiliser les instructions multiples (du type `INSERT … VALUES (…),(…),(…)`, etc., et `UPDATE` ou `DELETE` dont le `WHERE` ramène plusieurs lignes).
- Lorsque l'erreur survient dès la deuxième ligne, le comportement `STRICT_TRANS_TABLES` convertit la valeur problématique à la valeur de la colonne la plus proche acceptable. Si une valeur est manquante, une valeur implicite par défaut est substituée. Dans tous les cas, seul un *warning* est

généré et le traitement se poursuit.

Si vous n'adoptez pas un mode strict (ni `STRICT_TRANS_TABLES` ni `STRICT_ALL_TABLES`), MySQL ajustera au mieux les valeurs aux colonnes posant problème, tout en produisant des *warnings*. Ce comportement peut être explicitement mis en œuvre avec le mode en utilisant la directive `IGNORE` pour chaque `INSERT` et `UPDATE`.

Le mode strict n'a rien à voir avec les contraintes d'intégrité référentielles. C'est au niveau de la variable système `foreign_key_checks` (à 0 les contraintes sont désactivées, à 1 elles sont actives) qu'il faudra agir.

Les dates

Ce paragraphe concerne les colonnes `DATE` et `DATETIME`, et ne s'applique pas aux colonnes `TIMESTAMP` qui correspondent toujours à des dates valides.

Les modes stricts n'autorisent pas les dates invalides comme le 31 avril et les dates où l'année est différente de 0 avec un mois ou un jour nul. En revanche, ils laissent passer les dates ayant zéro dans l'année, quel que soit le mois ou le jour…

Pour débusquer certaines erreurs, vous devrez ajouter `NO_ZERO_IN_DATE` (qui interdit les jours ou les mois à zéro) et `NO_ZERO_DATE` (qui interdit `'0000-00-00'`) à votre mode strict.

Les modes `NO_ZERO_DATE` et `NO_ZERO_IN_DATE` sont déclarés obsolètes depuis la version 5.7 mais toutefois actifs par défaut. Ces modes seront inclus au mode SQL strict à moyen terme.

Si vous ne désirez opérer aucun contrôle mis à part le fait que le mois soit compris entre 1 et 12 et le jour entre 1 et 31, le mode `ALLOW_INVALID_DATES` peut vous intéresser. Combiné ou non à un mode strict, le contrôle complet ne sera pas réalisé (et le 31 avril devient valide). En revanche, en l'absence de mode, une date invalide se transforme en `'0000-00-00'` et un *warning* est généré.

Le tableau suivant présente un scénario illustrant ces différents cas d'utilisation.

Tableau 6-33 Modes SQL pour les dates

Commandes SQL	Commentaires

`SET SESSION sql_mode='STRICT_TRANS_TABLES';`
`CREATE TABLE Vols`
`  (num_vol CHAR(6), date_vol DATE,`
`   num_client INT, date_resa TIMESTAMP);`
	Initialisation du mode strict SQL.
	Création de la table `Vols`.

`INSERT INTO Vols (num_vol,date_vol,num_client)`
`  VALUES ('AF6143','2010-05-02',5);`
`INSERT INTO Vols (num_vol,date_vol,num_client)`
`  VALUES ('AF6143','0000-04-10',4);`
	Deux insertions valides (même si je doute qu'à l'année 0, il existait des aéronefs…)

`INSERT INTO Vols (num_vol,date_vol,num_client)` `  VALUES ('AF6143','2011-04-31',170);`	Insertion invalide : `ERROR 1292 (22007): Incorrect date value … for column 'date_vol'…`

`SET SESSION sql_mode=`
`  'STRICT_TRANS_TABLES,NO_ZERO_DATE';`
`INSERT INTO Vols (num_vol,date_vol,num_client)`
`  VALUES ('AF6143','0000-00-00',0);`
	Enrichissement du mode strict SQL. Insertion invalide (la date nulle est interdite du fait du mode SQL) : `ERROR 1292 (22007): Incorrect date value … for column 'date_vol'…`

`SET SESSION sql_mode=`
`  'STRICT_TRANS_TABLES,NO_ZERO_DATE,`
`  NO_ZERO_IN_DATE';`
`INSERT INTO Vols (num_vol,date_vol,num_client)`
`  VALUES ('AF6143','2011-04-00',70);`
`INSERT INTO Vols (num_vol,date_vol,num_client)`
`  VALUES ('AF6143','2011-00-10',165);`
`INSERT INTO Vols (num_vol,date_vol,num_client)`
`  VALUES ('AF6143','2011-00-00',1);`
	Enrichissement du mode strict SQL.
	Insertions invalides (le jour ou le mois nul est interdit du fait du mode SQL) : `ERROR 1292 (22007): Incorrect date value … for column 'date_vol'…`

`mysql> SELECT * FROM Vols;`
`+---------+------------+------------+---------------------+`
`| num_vol | date_vol  | num_client | date_resa           |`
`+---------+------------+------------+---------------------+`
`| AF6143  | 2010-05-02 |          5 | 2010-10-05 16:38:49 |`
`| AF6143  | 0000-04-10 |          4 | 2010-10-05 16:38:49 |`
`+---------+------------+------------+---------------------+`	

`SET SESSION sql_mode=`
`  'STRICT_TRANS_TABLES,ALLOW_INVALID_DATES';`
`INSERT INTO Vols (num_vol,date_vol,num_client)`
`  VALUES ('AF6143','2011-04-31',170);`
	Enrichissement du mode strict SQL.
	Insertion valide du fait du mode SQL.

`SET SESSION sql_mode='';`
`INSERT INTO Vols (num_vol,date_vol,num_client)`
`  VALUES ('AF6143','2012-04-31',367);`
	Annulation du mode strict SQL. Insertion valide (plus aucun contrôle). Un warning est toutefois généré : `1265 Data truncated for column`

```
                                                    'date_vol'...
```

```
mysql> SELECT * FROM Vols;
+---------+------------+------------+---------------------+
| num_vol | date_vol   | num_client | date_resa           |
+---------+------------+------------+---------------------+
| AF6143  | 2010-05-02 |          5 | 2010-10-05 16:38:49 |
| AF6143  | 0000-04-10 |          4 | 2010-10-05 16:38:49 |
| AF6143  | 2011-04-31 |        170 | 2010-10-05 17:36:38 |
| AF6143  | 0000-00-00 |        367 | 2010-10-05 17:36:38 |
+---------+------------+------------+---------------------+
```

Les séquences

Ce paragraphe concerne les colonnes AUTO_INCREMENT (séquence). Dans un mode strict ou par défaut, la valeur d'une séquence est générée séquentiellement même pour les colonnes dont la valeur est NULL ou 0 (voir chapitre 2, section « Séquences »).

Le mode NO_AUTO_VALUE_ON_ZERO inhibe ce comportement pour la valeur 0 qui ne déclenche pas une nouvelle séquence (alors que NULL le fait toujours). Ce mécanisme peut être intéressant lors d'un *dump* d'une table suivie d'un rechargement (la commande mysqldump utilise automatiquement ce mode).

Le tableau suivant présente quelques insertions illustrant ce mode d'utilisation.

Tableau 6-34 Mode SQL pour les séquences

Commandes SQL	Commentaires

`CREATE TABLE Affreter`
` (numAff SMALLINT **AUTO_INCREMENT**,`
`  comp CHAR(4),immat CHAR(6), dateAff DATE,`
`  nbPax SMALLINT(3),`
`  CONSTRAINT pk_Affreter PRIMARY KEY (numAff));`
	Création de la table.

`SET SESSION **sql_mode=''**;`
`INSERT INTO Affreter (comp,immat,dateAff,nbPax)`
`  VALUES ('AF', 'F-WTSS', '2012-05-13', 85);`
`INSERT INTO Affreter (numAff,comp,immat,dateAff,nbPax)`
`  VALUES (**NULL**, 'EJ','N-23DS', '2012-09-11', 90);`
`INSERT INTO Affreter (numAff,comp,immat,dateAff,nbPax)`
`  VALUES (**0**, 'AF','F-FPJY', '2012-09-11', 95);`
	Initialisation du mode vide (non strict). Insertion de trois lignes et séquences (les deux premières avec NULL, la troisième avec 0).

```
mysql> SELECT * FROM Affreter;
+--------+------+--------+------------+-------+
| numAff | comp | immat  | dateAff    | nbPax |
```

```
+--------+------+--------+------------+-------+
|      1 | AF   | F-WTSS | 2012-05-13 |    85 |
|      2 | EJ   | N-23DS | 2012-09-11 |    90 |
|      3 | AF   | F-FPJY | 2012-09-11 |    95 |
+--------+------+--------+------------+-------+
```

```
SET SESSION
sql_mode='NO_AUTO_VALUE_ON_ZERO';
INSERT INTO Affreter
(numAff,comp,immat,dateAff,nbPax)
  VALUES (NULL, 'AF','F-FRSS', '2012-09-
11', 127);
INSERT INTO Affreter
(numAff,comp,immat,dateAff,nbPax)
  VALUES (0, 'AF','F-GLFS', '2012-09-11',
75);
```

Enrichissement du mode SQL. Insertion de deux lignes et séquences (la première avec NULL, la deuxième avec 0).

```
mysql> SELECT * FROM Affreter ;
+--------+------+--------+------------+-------+
| numAff | comp | immat  | dateAff    | nbPax |
+--------+------+--------+------------+-------+
|      0 | AF   | F-GLFS | 2012-09-11 |    75 |
|      1 | AF   | F-WTSS | 2012-05-13 |    85 |
|      2 | EJ   | N-23DS | 2012-09-11 |    90 |
|      3 | AF   | F-FPJY | 2012-09-11 |    95 |
|      4 | AF   | F-FRSS | 2012-09-11 |   127 |
+--------+------+--------+------------+-------+
```

Chaînes de caractères

Dans tous les cas (mode strict ou non), les éventuels espaces qui suivent une valeur d'une colonne CHAR ne sont pas restitués à l'extraction (ces caractères sont en revanche présents dans la base). Le mode PAD_CHAR_TO_FULL_LENGTH inhibe ce comportement et ne restitue pas les valeurs de ces colonnes telles qu'elles sont stockées mais avec leur taille maximale. Ce mode ne s'applique pas aux colonnes VARCHAR pour lesquelles les valeurs sont toujours restituées telles qu'elles sont stockées.

Le tableau suivant présente quelques insertions illustrant ce mode d'utilisation.

Tableau 6-35 Mode SQL pour les chaînes de caractères

Commandes SQL	Commentaires

`CREATE TABLE Passager`
`(num_pax SMALLINT AUTO_INCREMENT PRIMARY KEY,`
`prenom CHAR(20), nom CHAR(20), mail VARCHAR(30));`
	Création de la table.

`SET sql_mode = '';`
`INSERT INTO Passager (prenom,nom,mail)`
`VALUES ('Fred','Brouard','sqlpro@sqlspot.com');`
	Annulation du mode strict. Insertion d'une ligne.

`SELECT CONCAT(prenom,LENGTH(prenom)) "Prenom",` `CONCAT(nom,LENGTH(nom)) "Nom",`	Les colonnes CHAR

```
       CONCAT(mail,LENGTH(mail)) "Mail"
  FROM Passager;
+--------+----------+---------------------+
| Prenom | Nom      | Mail                |
+--------+----------+---------------------+
| Fred4  | Brouard7 | sqlpro@sqlspot.com18 |
+--------+----------+---------------------+
```

sont restituées comme elles sont stockées.

```
SET sql_mode='PAD_CHAR_TO_FULL_LENGTH';
INSERT INTO Passager (prenom,nom,mail)
   VALUES ('Chris','Codd','coddy@dbms.com');
```

Enrichissement du mode SQL. Insertion de la deuxième ligne.

```
SELECT CONCAT(prenom,LENGTH(prenom)) "Prenom",
       CONCAT(nom,LENGTH(nom)) "Nom",
       CONCAT(mail,LENGTH(mail)) "Mail"
  FROM Passager;
+-----------------------+-----------------------+----------------------+
| Prenom                | Nom                   | Mail                 |
+-----------------------+-----------------------+----------------------+
| Fred               20 | Brouard            20 | sqlpro@sqlspot.com18 |
| Chris              20 | Codd               20 | coddy@dbms.com14     |
+-----------------------+-----------------------+----------------------+
```

Les moteurs

Il est possible d'affecter un moteur de stockage à une table lors de sa création par CREATE TABLE ou une fois créée par ALTER TABLE. Suivant votre configuration, ce moteur peut être indisponible (pas encore compilé ou désactivé). Le cas échéant, le mode NO_ENGINE_SUBSTITUTION provoque une erreur en retour et annule la création ou la modification de la table. Si ce mode n'est pas actif, la table est quand même créée avec le moteur de stockage par défaut (et un *warning* est retourné). En revanche, la table ne sera pas modifiée par ALTER TABLE et un *warning* sera également retourné.

Le tableau suivant présente quelques cas d'utilisation, dans lesquels on suppose que le moteur FEDERATED n'est pas implémenté. La première table est associée au moteur de stockage par défaut (ici InnoDB, voir votre variable default-storage-engine dans le fichier de configuration). La deuxième table n'est pas créée et une erreur est retournée.

Tableau 6-36 Mode SQL pour les moteurs

Création de tables

```
SET sql_mode = '';
CREATE TABLE Clients
 (num_cli SMALLINT AUTO_INCREMENT PRIMARY KEY,
  prenom CHAR(20), nom CHAR(20))  ENGINE=FEDERATED;
Query OK, 0 rows affected, 2 warnings (0.01 sec)

mysql> show warnings;
```

```
+---------+------+-------------------------------------------------+
| Level   | Code | Message                                         |
+---------+------+-------------------------------------------------+
| Warning | 1286 | Unknown storage engine 'FEDERATED'              |
| Warning | 1266 | Using storage engine InnoDB for table 'clients' |
+---------+------+-------------------------------------------------+
```

```
SET sql_mode = 'NO_ENGINE_SUBSTITUTION';
CREATE TABLE Avions
    (immat VARCHAR(8) PRIMARY KEY,
      typeav CHAR(20), mise_service DATE) ENGINE=FEDERATED;
ERROR 1286 (42000): Unknown storage engine 'FEDERATED'
```

Portabilité

Les modes NO_KEY_OPTIONS et NO_TABLE_OPTIONS masquent les options relatives aux tables spécifiques à MySQL en termes d'index, de commentaires et de moteur. Ces modes sont utilisés par la commande mysqldump. À titre d'exemple, supposons qu'un index soit défini sur la table Clients. Le tableau suivant présente l'utilisation de ces modes.

```
CREATE INDEX idx_nomcli USING BTREE
    ON Clients (nom DESC)
    COMMENT 'plus rapide pour les recherches nominatives';
```

Tableau 6-37 Modes SQL pour la portabilité

Sans le mode portabilité	Avec le mode portabilité

`SET sql_mode = '';`
`mysql> SHOW CREATE TABLE Clients;`
`+---------+-------------------------------`
`-`
`| Table   | Create Table`
`+---------+-------------------------------`
`-`
`| Clients | CREATE TABLE `clients` (`
`  `num_cli` smallint(6)`
`  NOT NULL AUTO_INCREMENT,`
`  `prenom` char(20) DEFAULT NULL,`
`  `nom` char(20) DEFAULT NULL,`
`  PRIMARY KEY (`num_cli`),`
`  KEY `idx_nomcli` (`nom`)`
`  USING BTREE`
`  COMMENT 'plus rapide pour les`
`  recherches`
`nominatives')`
`  ENGINE=InnoDB DEFAULT`
`  CHARSET=latin1`
	`SET sql_mode = 'NO_KEY_OPTIONS,`
	`  NO_TABLE_OPTIONS';`
	`mysql> SHOW CREATE TABLE Clients;`
	`+---------+-----------------------------`
	`| Table   | Create Table`
	`+---------+-----------------------------`
	`| Clients | CREATE TABLE `clients` (`
	`  `num_cli` smallint(6)`
	`  NOT NULL AUTO_INCREMENT,`
	`  `prenom` char(20) DEFAULT`
	`NULL,`
	`  `nom` char(20) DEFAULT NULL,`
	`  PRIMARY KEY (`num_cli`),`
	`  KEY `idx_nomcli` (`nom`))`

Le mode IGNORE_SPACE permet de placer un espace entre le nom d'une fonction MySQL (ne s'applique pas aux fonctions cataloguées) et les paramètres (devant le caractère « (»). Le nom d'une fonction est alors considéré comme un mot

réservé du vocable MySQL. Il n'est jamais possible, quel que soit le mode SQL, de séparer le nom d'une fonction utilisateur (ou d'une fonction stockée) de ses paramètres.

Tableau 6-38 Mode SQL IGNORE_SPACE

Sans le mode IGNORE_SPACE	Avec le mode IGNORE_SPACE
mysql> SET sql_mode = ''; mysql> SELECT MAX(nom) FROM Passager; +----------+ | MAX(nom) | +----------+ | Codd | +----------+ mysql> SELECT MIN (nom) FROM Passager; ERROR **1630** (42000): FUNCTION bdsoutou.MIN does not exist…	mysql> SET sql_mode = '**IGNORE_SPACE**'; mysql> SELECT MAX(nom), MIN (nom) FROM Passager; +----------+----------+ | MAX(nom) | MIN (nom) | +----------+----------+ | Codd | Brouard | +----------+----------+

Les combinaisons

Certains modes sont composés de plusieurs modes élémentaires, parmi lesquels nous pouvons citer :

- ANSI combine REAL_AS_FLOAT (considère les colonnes REAL comme FLOAT et non comme DOUBLE), PIPES_AS_CONCAT (|| permet de concaténer deux chaînes de caractères), ANSI_QUOTES (considère les caractères " et ` pour délimiter un identifiant et non une chaîne de caractères) et IGNORE_SPACE. Ce mode se rapproche le plus du standard SQL.
- TRADITIONAL combine STRICT_TRANS_TABLES, STRICT_ALL_TABLES, NO_ZERO_IN_DATE, NO_ZERO_DATE, ERROR_FOR_DIVISION_BY_ZERO, NO_AUTO_CREATE_USER et NO_ENGINE_SUBSTITUTION. Ce mode ajoute à un mode strict des restrictions sur les données en entrée.
- ORACLE combine PIPES_AS_CONCAT, ANSI_QUOTES, IGNORE_SPACE, NO_KEY_OPTIONS, NO_TABLE_OPTIONS, NO_FIELD_OPTIONS et NO_AUTO_CREATE_USER. Ce mode se rapproche le plus de la syntaxe d'Oracle.
- POSTGRESQL combine PIPES_AS_CONCAT, ANSI_QUOTES, IGNORE_SPACE, NO_KEY_OPTIONS, NO_TABLE_OPTIONS et NO_FIELD_OPTIONS. Ce mode se rapproche le plus de la syntaxe PostgreSQL.
- DB2 combine PIPES_AS_CONCAT, ANSI_QUOTES, IGNORE_SPACE, NO_KEY_OPTIONS, NO_TABLE_OPTIONS et NO_FIELD_OPTIONS. Ce mode se rapproche le plus de la syntaxe d'IBM DB2.

- MSSQL combine PIPES_AS_CONCAT, ANSI_QUOTES, IGNORE_SPACE, NO_KEY_OPTIONS, NO_TABLE_OPTIONS et NO_FIELD_OPTIONS. Ce mode se rapproche le plus de la syntaxe de Microsoft SQL Server.

Prudence si vous utilisez un tel mode en exécutant des instructions dont la syntaxe est propriétaire MySQL (les énumérations ENUM et SET par exemple).

Exercices

L'objectif de ces exercices est d'écrire des blocs puis des transactions manipulant des tables du schéma *Parc Informatique*. Vous utiliserez une procédure pour tester vos blocs, comme il est indiqué dans la section « Test des exemples ».

Exercice 6.1

Extraction de données

Écrire le bloc MySQL qui affiche les détails de la dernière installation de logiciel sous la forme suivante (les champs en gras sont à extraire) :

```
+-----------------------------------------------+
| Resultat 1 exo 1                              |
+-----------------------------------------------+
| Derniere installation en salle : numérodeSalle |
+-----------------------------------------------+
+-------------------------------------------------------------------+
| Resultat 2 exo 1                                                  |
+-------------------------------------------------------------------+
| Poste : numéroPoste Logiciel : nomLogiciel en date du dateInstallation |
+-------------------------------------------------------------------+
```

Vous utiliserez SELECT … INTO pour extraire ces valeurs. Ne tenez pas compte, pour le moment, des erreurs qui pourraient éventuellement se produire (aucune installation de logiciel, poste ou logiciel non référencés dans la base, etc.).

Exercice 6.2

Variables de session

Écrire le bloc MySQL qui affecte hors d'un bloc, par des variables session, un numéro de salle et un type de poste, et qui retourne des variables session

permettant de composer un message indiquant les nombres de postes et d'installations de logiciels correspondants :

```
+-------------------------------------------------------------------+

| Resultat exo2                                                     |
+-------------------------------------------------------------------+
| x poste(s) installe(s) en salle y, z installation(s) de type t    |
+-------------------------------------------------------------------+
```

Essayez pour la salle `s01` et le type `UNIX`. Vous devez extraire 1 poste et 3 installations. Ne tenez pas compte pour le moment d'éventuelles erreurs (aucun poste trouvé ou aucune installation réalisée, etc.).

Exercice 6.3

Transaction

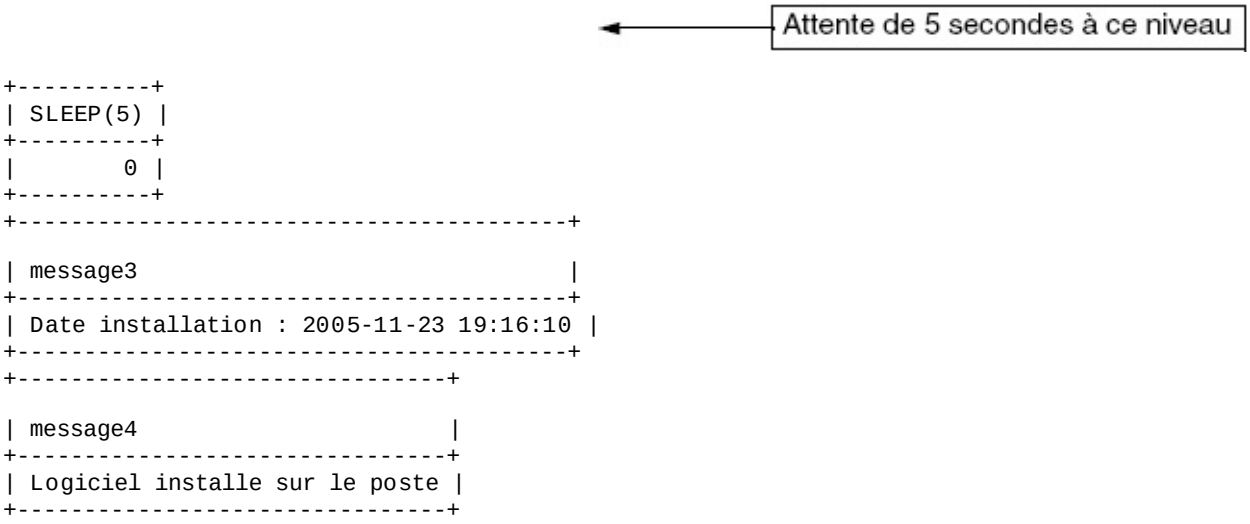
Écrire une transaction permettant d'insérer un nouveau logiciel dans la base après avoir passé en paramètres, par des variables de session, toutes ses caractéristiques (numéro, nom, version et type du logiciel). La date d'achat doit être celle du jour. Tracer l'insertion du logiciel (message `Logiciel` inséré dans la base).

Il faut ensuite procéder à l'installation de ce logiciel sur le poste de code `p7` (utiliser une variable pour pouvoir plus facilement modifier ce paramètre). L'installation doit se faire aussi à la date du jour. Penser à actualiser correctement la colonne `delai` qui mesure le délai (`TIME`) entre l'achat et l'installation. Pour ne pas que ce délai soit nul (les deux insertions se feraient dans la même seconde dans cette transaction), placer une attente de 5 secondes entre l'ajout dans la table `Logiciel` et celui dans la table `Installer` à l'aide de l'instruction `SELECT SLEEP(5)`. Utiliser la fonction `TIMEDIFF` pour calculer ce délai.

Insérer par exemple le logiciel `log15`, de nom `MySQL Query`, version `1.4`, `typePCWS` coûtant 95 €. Tracer la transaction comme suit :

```
+-----------------------------+
| message1                    |
+-----------------------------+
| Logiciel insere dans la base |
+-----------------------------+
1 row in set (0.01 sec)
+---------------------------------+

| message2                        |
+---------------------------------+
| Date achat : 2005-11-23 19:16:04 |
+---------------------------------+
```



Vérifiez l’état des tables mises à jour après la transaction. Ne tenez pas compte pour le moment d’éventuelles erreurs (numéro du logiciel déjà référencé, type du logiciel incorrect, installation déjà réalisée, etc.).

Chapitre 7

Programmation avancée

Ce chapitre est consacré à des caractéristiques avancées du langage procédural de MySQL (sous-programmes, curseurs, exceptions, déclencheurs, SQL dynamique, événements et gestion de documents XML et JSON).

Sous-programmes

Les sous-programmes sont des blocs nommés qui sont compilés et qui résident dans la base de données. Dans le vocabulaire des bases de données, on appelle les sous-programmes *stored procedures* ou *stored routines*. Ce sont des fonctions ou procédures « cataloguées » (ou « stockées ») capables d'inclure des paramètres en entrée. Comme dans tous les langages de programmation, les fonctions retournent un unique résultat, alors que les procédures réalisent des actions sans en donner (sauf éventuellement en paramètre de sortie).

Étant un des plus « jeunes » des SGBD, MySQL tend au plus près (en ajoutant toutefois des extensions) de la syntaxe normative de SQL:2003 (sections « Stored Modules » et « Computational completeness »). L'autre SGBD se rapprochant le plus de la norme est DB2 d'IBM. Oracle et SQL Server de Microsoft ont un grand nombre de caractéristiques absentes de la norme.

Pour l'instant, seules les procédures sont capables d'inclure des paramètres en sortie.

Un sous-programme ne se recompile pas automatiquement suite à la modification d'un objet de la base manipulé dans son code (ajout d'une colonne dans une table par exemple).

Généralités

Il est possible de retrouver le code d'un sous-programme au niveau du dictionnaire des données (voir la fin du chapitre 5). Le sous-programme peut être ainsi partagé dans un contexte multi-utilisateur. Les avantages d'utiliser des sous-programmes ont été soulignés au chapitre 6 (modularité, portabilité, extensibilité, réutilisabilité, intégrité et confidentialité).

Comme les blocs, nous verrons que les sous-programmes ont une partie de déclaration des variables, une autre contenant les instructions et éventuellement une partie pour gérer les exceptions (erreurs produites durant l'exécution).

Une procédure peut être appelée à l'aide de l'interface de commande (par CALL), dans un programme externe (Java, PHP, C…), par d'autres procédures ou fonctions, ou dans le corps d'un déclencheur. Les fonctions peuvent être invoquées dans une instruction SQL (SELECT, INSERT, et UPDATE) ou dans une expression (affectation de variable ou calcul).

Le cycle de vie d'un sous-programme est le suivant : création de la procédure ou de la fonction (compilation et stockage dans la base), appel et éventuellement suppression du sous-programme de la base.

Procédures cataloguées

La syntaxe de création d'une procédure cataloguée est la suivante. Le privilège CREATE ROUTINE est requis sur la base de données (ou au niveau global) en question (ALTER ROUTINE et EXECUTE sont affectés par la suite automatiquement).

```
CREATE PROCEDURE [nomBase.]nomProcédure(
                [ [ IN | OUT | INOUT ] param typeMySQL
                [,[ IN | OUT | INOUT ] param2 typeMySQL ] ] ...)
    [ LANGUAGE SQL
      | [NOT] DETERMINISTIC
      | { CONTAINS SQL | NO SQL | READS SQL DATA | MODIFIES SQL DATA }
      | SQL SECURITY { DEFINER | INVOKER }
      | COMMENT 'commentaire'
    ]
BEGIN
 [DECLARE ... ;]
bloc d'instructions SQL et MySQL ... ;
END;
délimiteur
```

- Par défaut, la procédure est créée dans la base de données courante (sélectionnée). Si un nom est spécifié (nomBase), la procédure appartiendra à

cette base de données.

- `IN` désigne un paramètre d'entrée (par défaut), `OUT` un paramètre de sortie et `INOUT` un paramètre d'entrée et de sortie.
- `LANGUAGE SQL` (par défaut) détermine le langage de programmation de la procédure. MySQL n'est pas encore compatible avec d'autres langages que le sien.
- `DETERMINISTIC` est simplement informationnel (l'optimiseur s'en servira dans des versions ultérieures) et décrit le caractère déterministe de la procédure. Si vous interrogez la base, il serait plus naturel d'utiliser `NOT DETERMINISTIC`, car on ne sait pas a priori ce que l'on va extraire (comme dans la boîte de chocolats de *Forrest Gump)*.
- `CONTAINS SQL` renseigne sur le fait que la procédure interagit avec la base. `NO SQL` indique l'inverse. `READS SQL DATA` précise que les interactions sont en lecture seulement. `MODIFIES SQL DATA` signifie que des mises à jour de la base sont possibles.
- `SQL SECURITY` détermine si la procédure s'exécute avec les privilèges du créateur (option par défaut : *definer-rights procedure*) ou ceux de l'utilisateur qui appelle la procédure (*invoker-rights procedure*).
- `COMMENT` permet de commenter la procédure au niveau du dictionnaire des données (voir chapitre 5).
- `bloc d'instructions SQL et MySQL` contient les déclarations et les instructions de la procédure écrite dans le langage de MySQL (voir le chapitre précédent).
- `délimiteur` : délimiteur de commandes différent de « ; » (symbole utilisé obligatoirement en fin de chaque déclaration et instruction du langage procédural de MySQL).

Fonctions cataloguées

La syntaxe de création d'une fonction cataloguée est `CREATE FUNCTION`. Les prérogatives et les options sont les mêmes que pour les procédures. N'oubliez pas l'instruction « `RETURN variable;` » qui termine la fonction et retourne le résultat (de même type que celui déclaré dans la clause `RETURNS`.

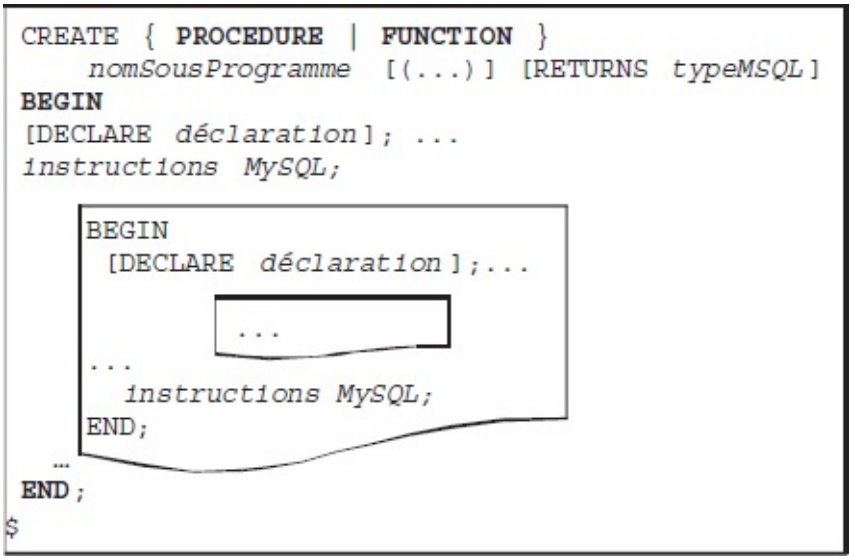
```
CREATE FUNCTION [nomBase.]nomFonction(
                        [ param typeMySQL
                        [,param2 typeMySQL ] ] ...)
    RETURNS typeMySQL
```

```
    [ LANGUAGE SQL
      | [NOT] DETERMINISTIC
      | { CONTAINS SQL | NO SQL | READS SQL DATA | MODIFIES SQL DATA }
      | SQL SECURITY { DEFINER | INVOKER }
      | COMMENT 'commentaire'
    ]
BEGIN
 [DECLARE ... ;]
bloc d'instructions SQL et MySQL ... ;
contenant un « RETURN variable ; »
END;
délimiteur
```

Structure d'un sous-programme

Dans une procédure, comme dans une fonction, les déclarations des variables, curseurs et exceptions suivent directement l'en-tête du bloc (après la directive `BEGIN`). La figure suivante illustre la structure d'une spécification et d'un corps d'un sous-programme MySQL. Le bloc d'instructions doit contenir au moins une instruction MySQL.

Figure 7-1 *Structure d'un sous-programme*



Exemples

Considérons la table `Pilote`. Nous allons écrire (dans la base `bdsoutou`) une fonction et une procédure :

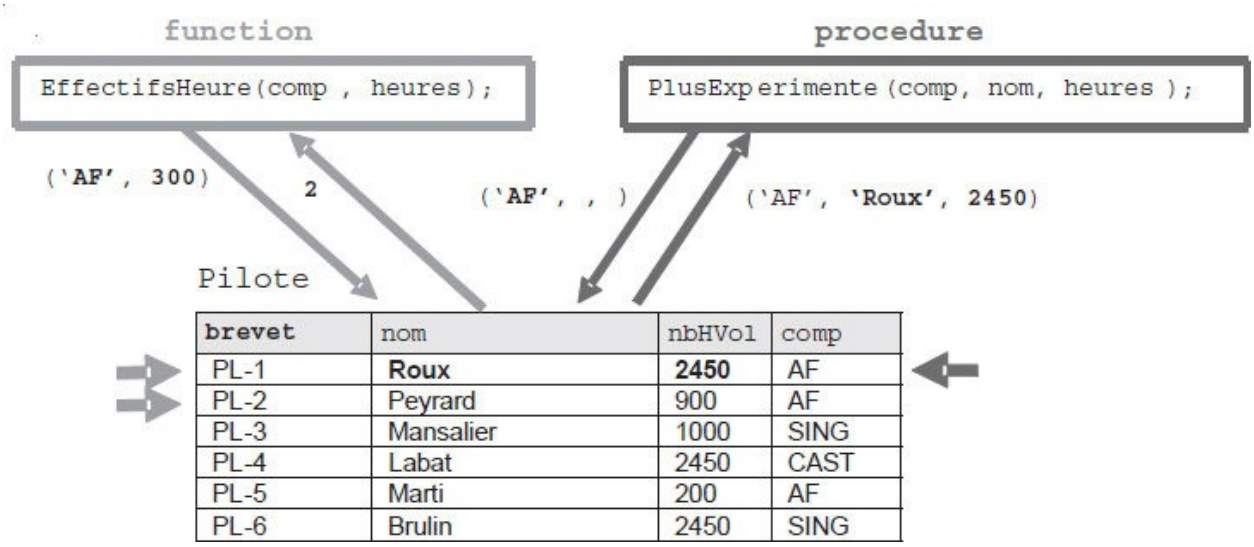
- La fonction `EffectifsHeure(comp,heures)` devra renvoyer le nombre de pilotes d'une compagnie donnée (premier paramètre) qui ont plus d'heures de vol

que la valeur du deuxième paramètre (si aucun pilote, retourne 0). Si aucune compagnie n'est passée en paramètre (mettre `NULL`), le calcul inclut toutes les compagnies. Les éventuelles erreurs ne sont pas encore traitées (compagnie de code inexistant, par exemple).

- La procédure `PlusExperimente(comp,nom,heures)` doit retourner le nom et le nombre d'heures de vol du pilote (par l'intermédiaire des deuxième et troisième paramètres) le plus expérimenté d'une compagnie donnée (premier paramètre). Si plusieurs pilotes ont la même expérience, un message d'erreur est affiché. Si aucune compagnie n'est passée en paramètre (mettre `NULL`), la procédure retourne le nom du plus expérimenté et le code de sa compagnie (par l'intermédiaire du premier paramètre).

Remarquez que la fonction aurait pu être programmée par une procédure ayant un troisième paramètre de sortie.

Figure 7-2 *Procédures*



Une fonction

La création de la fonction est réalisée à l'aide du script suivant (`EffectifsHeure.sql`). Notez que les deux paramètres d'entrée ne sont pas définis par la directive `IN` et que la clause `RETURN` doit être présente en fin de codage.

```
CREATE FUNCTION bdsoutou.EffectifsHeure(pcomp VARCHAR(4),
                    pheuresVol DECIMAL(7,2)) RETURNS SMALLINT
BEGIN
 DECLARE resultat SMALLINT;
 IF (pcomp IS NULL) THEN
    SELECT COUNT(*) INTO resultat FROM Pilote WHERE nbHVol > pheures-
```

```
Vol;
 ELSE
    SELECT COUNT(*) INTO resultat FROM Pilote WHERE nbHVol > pheuresVol
                    AND comp = pcomp;
 END IF;
RETURN resultat;
END;
```

Une procédure

La création de la procédure est réalisée à l'aide du script suivant (`PlusExperimente.sql`). Notez les deux derniers paramètres de sortie définis par la directive `OUT`, et le premier servant d'entrée ou de sortie avec la directive `INOUT`. On peut assimiler le passage d'un paramètre par référence à l'utilisation de la directive `INOUT`.

```
CREATE PROCEDURE bdsoutou.PlusExperimente


(INOUT pcomp VARCHAR(4), OUT pnomPil VARCHAR(20), OUT pheuresVol DECI
BEGIN
 DECLARE p1 SMALLINT;
 IF (pcomp IS NULL) THEN
   SELECT COUNT(*) INTO p1 FROM Pilote
          WHERE nbHVol=(SELECT MAX(nbHVol) FROM Pilote);
 ELSE
   SELECT COUNT(*) INTO p1 FROM Pilote
           WHERE nbHVol=(SELECT MAX(nbHVol) FROM Pilote WHERE
comp=pcomp)
          AND comp = pcomp;
 END IF;
 IF (p1 = 0) THEN
   SELECT ('Aucun pilote n''est le plus expérimenté') AS resultat;
 ELSEIF p1 > 1 THEN
   SELECT('Plusieurs pilotes sont les plus expérimentés') AS resultat;
 ELSE
   IF (pcomp IS NULL) THEN
       SELECT nom, nbHVol, comp INTO pnomPil, pheuresVol, pcomp
         FROM Pilote WHERE nbHVol=(SELECT MAX(nbHVol) FROM Pilote);
   ELSE
       SELECT nom, nbHVol INTO pnomPil, pheuresVol FROM Pilote
            WHERE nbHVol=(SELECT MAX(nbHVol) FROM Pilote WHERE
comp=pcomp)
           AND comp = pcomp;
   END IF;
 END IF;
END;
```

Fonction n'interagissant pas avec la base

La fonction `EcritureComplexe` renvoie une chaîne de caractères désignant l'écriture d'un nombre complexe sous la forme « *a* + *b*i » ou « *a* - *b*i » (`EcritureComplexe.sql`), en fonction du signe des deux paramètres *a* et *b* définissant

la partie réelle et la partie imaginaire du complexe.

```
CREATE FUNCTION bdsoutou.EcritureComplexe
               (reel DECIMAL(7,2), imaginaire DECIMAL(7,2))
                RETURNS VARCHAR(80)
BEGIN
 DECLARE result VARCHAR(80);
 IF (imaginaire < 0) THEN
   SET result := CONCAT('Complexe : ',reel,'-',-imaginaire,'i');
 ELSE
    SET result := CONCAT('Complexe : ',reel,'+',imaginaire,'i');
 END IF;
RETURN result ;
END;
```

Compilation

Pour compiler ces sous-programmes à partir de l'interface de commande, il faut ajouter un délimiteur après chaque dernier `END` comme suit :

```
delimiter $
Sous programme ;
$
```

Si un message d'erreur apparaît, il indique la ligne concernée ; souvent l'erreur peut être située juste avant cette ligne. Le même résultat peut être obtenu par la commande `SHOW ERRORS`.

Une fois que le message « `Query OK, 0 rows affected (… sec)` » apparaît, le sous-programme est correctement compilé.

Appel d'un sous-programme

Le créateur d'un sous-programme peut exécuter ce dernier à la demande et sans aucune condition préalable. Pour exécuter un sous-programme à partir d'un autre accès, les conditions suivantes doivent être respectées :

- détenir le privilège `EXECUTE` sur la procédure en question ou sur la base qui contient le sous-programme, ou au niveau global ;
- mentionner le nom de la base (du schéma) contenant le sous-programme à l'exécution de ce dernier (exemple d'appel sous l'interface de commande de la procédure `AugmenteCapacite` de la base `bdjean`, pour l'avion d'immatriculation `'F-GLFS'` : « `CALL bdjean.AugmenteCapacite('F-GLFS');` »).

Décrivons l'appel d'un sous-programme sous l'interface de commande, dans

un sous-programme MySQL et dans une instruction SQL. Le chapitre suivant détaillera un tel appel à partir d'un programme externe (Java ou PHP).

Sous l'interface de commande

En phase de tests, il est intéressant de pouvoir déclencher un sous-programme directement dans l'interface de commande. La commande CALL permet d'appeler une procédure. Une fonction est exécutée par son nom dans une instruction SQL.

Le tableau suivant décrit l'appel et le résultat des trois sous-programmes. La fonction est appelée ici dans un SELECT et dans un INSERT de deux manières différentes.

Tableau 7-1 Appels dans l'interface de commande

Appel d'un sous-programme	Résultat

`delimiter ;`
`SELECT bdsoutou.EffectifsHeure`
`  ('AF',300);`
	`+----------------------------------+`
	`| bdsoutou.EffectifsHeure('AF',300) |`
	`+----------------------------------+`
	`|                                2 |`
	`+----------------------------------+`

`delimiter ;`
`SET @vs_compa = 'AF';`
`SET @vs_nompil = '';`
`SET @vs_heures = '';`
 	
`CALL bdsoutou.PlusExperimente`
`(@vs_compa, @vs_nompil,`
`@vs_heures);`
	`SELECT @vs_compa,@vs_nompil,@vs_heures;`
	`+----------+---------------+------------+`
	`| @vs_compa | @vs_nompil  | @vs_heures`
	`|`
	`+----------+---------------+------------+`
	`| AF  | Roux  |`
	`2450.00  |`
	`+----------+---------------+------------+`

`delimiter ;`
`SELECT`
`bdsoutou.EcritureComplexe(2,-5);`
	`+--------------------------------+`
	`| bdsoutou.EcritureComplexe(2,-5) |`
	`+--------------------------------+`
	`| Complexe : 2-5i  |`
	`+--------------------------------+`

`CREATE TABLE test.Trace`
`  (col VARCHAR(80));`
`INSERT INTO test.Trace`
`  SELECT bdsoutou.EcritureComplexe`
`bdsoutou.EcritureComplexe`
`  (-2,-5);`
`INSERT INTO test.Trace VALUES`
`bdsoutou.EcritureComplexe(3,-7));`
	`SELECT * FROM test.Trace;`
	`+------------------+`
	`| col  |`
	`+------------------+`
	`| Complexe : -2-5i |`
	`| Complexe : 3-7i  |`
	`+------------------+`

Dans un sous-programme

Invoquons les sous-programmes à présent à partir d'un autre sous-

programme. Le même principe peut être adopté pour l'appel dans un déclencheur. La procédure s'appelle toujours par CALL, la fonction par son nom (ici elle est appelée dans l'affectation d'une variable).

Tableau 7-2 Appels dans un sous-programme

Codage	Appels

`CREATE PROCEDURE test.sp1()`
`BEGIN`
` DECLARE v_compa  VARCHAR(4)`
`  DEFAULT 'AF';`
` DECLARE v_heures DECIMAL(7,2)`
`  DEFAULT 300;`
` DECLARE v_nbpil  SMALLINT;`
` SET v_nbpil := bdsoutou.EffectifsHeure`
`  (v_compa,v_heures);`
` SELECT v_nbpil;`
`END;`
	`CALL test.sp1()$`
	`+---------+`
	`| v_nbpil |`
	`+---------+`
	`|  2 |`
	`+---------+`

`SET @vs_compa  = NULL$`
`SET @vs_nompil = ''$`
`SET @vs_heures = ''$`
`CREATE PROCEDURE test.sp2()`
`BEGIN`
` CALL bdsoutou.PlusExperimente`
`  (@vs_compa,@vs_nompil,@vs_heures);`
`END;`
	`CALL test.sp2()$`
	`+---------------------------------+`
	`| resultat  |`
	`+---------------------------------+`
	`| Plusieurs pilotes sont les plus`
	`  expérimentés  |`
	`+---------------------------------+`
	`-- Les variables de session sont`
	`-- toutes à NULL`

`SET @vs_resultat  = ''$`
`CREATE PROCEDURE test.sp3()`
`BEGIN`
` SET @vs_resultat :=`
`  bdsoutou.EcritureComplexe(7,-2);`
`END;`
	`CALL test.sp3()$`
	`SELECT  @vs_resultat AS 'Résultat'$`
	`+-----------------+`
	`| Résultat  |`
	`+-----------------+`
	`| Complexe : 7-2i |`
	`+-----------------+`

Récursivité

La récursivité n'est pour l'instant pas permise dans MySQL au niveau des sous-programmes.

Comme dans tout programme récursif, il ne faudrait pas oublier la condition de terminaison ! L'exemple suivant décrit la programmation à l'aide d'une fonction récursive du calcul de la factorielle d'un entier positif. La compilation se déroule sans erreur, l'appel, lui, nous ramène à l'ordre.

Tableau 7-3 Récursivité

Code MYSQL	Commentaires

delimiter $
CREATE FUNCTION **factorielle**(n INT)
RETURNS INT
BEGIN
IF n = 1 THEN
RETURN (1);
ELSE
RETURN (n * **factorielle**(n - 1));
END IF;
END;
$;
	Condition de terminaison.
	Appel récursif.

SELECT AS 'Factorielle de 10'$ factorielle(10) ERROR **1424** (HY000): Recursive stored functions and triggers are not allowed.	Appel de la fonction !

Sous-programmes imbriqués

Il n'est pas possible de créer un sous-programme (*nested subprogram*) dans un autre sous-programme.

Cela n'est valable que pour les blocs d'instructions qui peuvent éventuellement en inclure d'autres (voir chapitre 6).

Le tableau suivant décrit la déclaration invalide du sous-programme `Mouchard` dans la procédure `imbriquee`. Ce sous-programme insérerait une ligne dans une table pour tracer l'appel de la procédure en fonction de l'utilisateur et du moment de l'exécution.

Tableau 7-4 Sous-programme imbriqué

Code MySQL	Commentaires

CREATE PROCEDURE bdsoutou.imbriquee
(INOUT p1 VARCHAR(2))
BEGIN
CREATE PROCEDURE bdsoutou.**Mouchard**()
BEGIN
INSERT INTO test.Trace VALUES
(CONCAT(USER(),' a lancé imbriquee
le ',SYSDATE()));
END;
	Déclaration du sous-programme.
	Déclaration du sous-programme imbriqué.

```
  SET p1 := 'OK';
  CALL bdsoutou.Mouchard();
END;
$
```

Début du sous-programme.

Appel du sous-programme imbriqué.

La compilation renvoie un message d'erreur très clair :

```
ERROR 1303 (2F003): Can't create a PROCEDURE from within another stored routine
```

Deux solutions s'offrent à vous : vous séparez les deux procédures et en appelez une par `CALL` dans l'autre, ou vous incluez la procédure appelée à l'intérieur d'un bloc `BEGIN`… `END` dans la procédure de plus haut niveau.

Modification d'un sous-programme

La modification d'un sous-programme s'exécute par la commande `ALTER`. Pour pouvoir changer un sous-programme, si vous n'êtes pas son créateur, vous devez détenir le privilège `ALTER ROUTINE` sur la base de données (ou au niveau global). Plusieurs caractéristiques peuvent être corrigées en une seule instruction (mais ni le code, ni les paramètres). La syntaxe générale est la suivante :

```
ALTER {PROCEDURE | FUNCTION} [nomBase.]nomSousProg
    [ { CONTAINS SQL | NO SQL | READS SQL DATA | MODIFIES SQL DATA }
      | SQL SECURITY { DEFINER | INVOKER }
      | COMMENT 'commentaire'
    ...]
```

L'instruction suivante commente le sous-programme `Mouchard` présent dans la base `bdsoutou` en indiquant qu'il interagit avec la base en écriture, et qu'il s'exécutera avec les privilèges de l'accès utilisateur qui l'a créé.

```
ALTER PROCEDURE bdsoutou.Mouchard
      MODIFIES SQL DATA
      SQL SECURITY DEFINER
      COMMENT 'Traces de qui lance quoi';
```

Destruction d'un sous-programme

Pour supprimer un sous-programme, si vous n'êtes pas son créateur, le privilège `ALTER ROUTINE` est requis sur la base de données (ou au niveau global). La syntaxe de suppression d'un sous-programme est la suivante :

```
DROP {PROCEDURE | FUNCTION} [IF EXISTS] [nomBase.]nomSousProg
```

- `IF EXISTS` évite un message de *warning* si le sous-programme n'existe pas.

Les instructions suivantes suppriment la procédure cataloguée `Mouchard` de la base `bdsoutou`.

```
delimiter ;
DROP PROCEDURE bdsoutou.Mouchard;
```

Restrictions

Les restrictions que nous mentionnons ici s'appliquent également aux déclencheurs (étudiés en fin de ce chapitre). Bien qu'il existe des restrictions qui s'appliquent seulement aux fonctions et aux déclencheurs, elles peuvent s'appliquer à une procédure si cette dernière est appelée dans le code de la fonction ou du déclencheur.

- Les instructions suivantes ne peuvent être présentes dans un sous-programme `CHECK TABLES`, `LOCK TABLES`, `UNLOCK TABLES`, `LOAD DATA`, `LOAD TABLE` et `OPTIMIZE TABLE`.
- Il n'est pas possible de programmer des instructions SQL en dynamique (`PREPARE`, `EXECUTE`, `DEALLOCATE PREPARE`) dans un déclencheur (possible dans une fonction ou une procédure).
- Il n'y a pas encore d'outil de débogage pour les sous-programmes.
- Les instructions `CALL` ne peuvent être préparées à l'avance (`CALL variable`).
- MySQL ne prend pas encore en charge la notion de paquetage (*package*) qui est un module regroupant plusieurs objets (variables, exceptions, curseurs, fonctions ou procédures) fournissant un ensemble de services (un peu comme une classe dans l'approche objet).

Curseurs

Le mécanisme de curseurs permet une programmation itérative alors que le

langage SQL préconise une programmation ensembliste. Il doit être utilisé à bon escient, c'est-à-dire lorsqu'il n'existe pas d'autre solution d'implémentation sous la forme d'une ou de plusieurs instructions SQL.

Les curseurs de MySQL peuvent être utilisés à l'intérieur d'un sous-programme (procédure ou fonction cataloguée) ou dans un déclencheur. Le concept analogue est appelé différemment selon les langages : `Resultset` avec Java (JDBC), `RecordSet` et `DataSet` avec Microsoft.

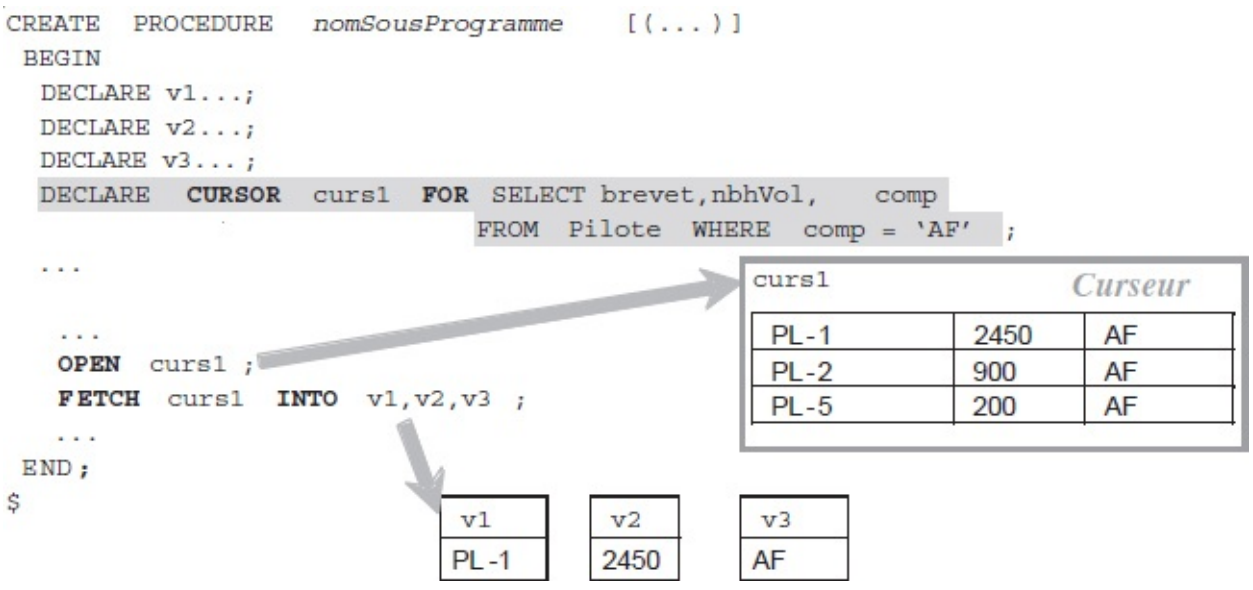
Généralités

Un curseur est une zone mémoire qui est générée côté serveur (mise en cache) et qui permet de traiter individuellement chaque ligne renvoyée par un SELECT. Un sous-programme peut travailler avec plusieurs curseurs en même temps. Un curseur, durant son existence (de l'ouverture à la fermeture), contient en permanence l'adresse de la ligne courante.

La figure suivante illustre la manipulation de base d'un curseur. Le curseur est décrit après les variables. Il est ouvert dans le code du programme ; il s'évalue alors et va se charger en extrayant les données de la base. Le programme peut parcourir tout le curseur en récupérant les lignes une par une dans une variable locale. Le curseur est ensuite fermé.

Les curseurs doivent être déclarés après les variables et avant les exceptions.

***Figure 7-3** Principes d'un curseur*



Tout curseur MySQL dispose des propriétés suivantes :

- lecture seule : aucune modification dans la base n'est possible à travers ce dernier ;
- non navigable : une fois ouvert, le curseur est parcouru du début à la fin sans pouvoir revenir à l'enregistrement précédent ;
- insensible (*asensitive*) : toute mise à jour opérée dans la base n'est pas répercutée dans le curseur une fois ouvert.

Instructions

Les instructions disponibles pour travailler avec des curseurs sont définies dans le tableau suivant :

Tableau 7-5 Instructions pour les curseurs

Instructions	Commentaires et exemples
CURSOR FOR *requête*;	Déclaration du curseur. `DECLARE curs1 CURSOR FOR SELECT brevet,nbHVol,comp FROM bdsoutou.Pilote WHERE comp = 'AF';`
OPEN *nomCurseur*;	Ouverture du curseur (chargement des lignes). Aucune exception n'est levée si la requête ne ramène aucune ligne. `OPEN curs1;`
FETCH *nomCurseur* **INTO** *listeVariables*;	Positionnement sur la ligne suivante et chargement de l'enregistrement courant dans une ou plusieurs variables. `FETCH curs1 INTO var1,var2,var3;`
CLOSE *nomCurseur*;	Ferme le curseur. L'exception « `ERROR 1326 (24000): Cursor is not open` » se déclenche si des accès au curseur sont opérés après sa fermeture. `CLOSE curs1;`

Parcours d'un curseur

Le parcours d'un curseur nécessite une des structures répétitives (*tant que*, *répéter* ou boucle *loop*). La boucle *loop* semble être la plus employée.

Le tableau suivant présente le parcours d'un curseur à l'aide de ces deux techniques. Ici, il s'agit de faire la somme des heures de vol des pilotes de la compagnie ayant pour code '`AF`'.

Notez l'utilisation obligatoire d'une exception (*handler* dans le vocable de MySQL) qui force le programme à continuer si on arrive en fin du curseur (au-delà du dernier enregistrement), tout en positionnant la variable `curs1` à *vrai* (1). Je dis « obligatoire », car MySQL ne propose pas, pour l'heure, de fonctions sur les curseurs (`FOUND`, `NOT_FOUND`, `IS_CLOSED`, etc.). Nous étudierons plus en détail les exceptions dans la prochaine section.

Tableau 7-6 Parcours d'un curseur

Tant que	Répéter

`DECLARE fincurs1 BOOLEAN DEFAULT 0;`
`DECLARE v_nbHv DECIMAL(7,2);`
`DECLARE v_tot DECIMAL(8,2) DEFAULT 0;`
`DECLARE curs1 CURSOR FOR`
`SELECT nbHVol FROM bdsoutou.Pilote WHERE comp = 'AF';`
`DECLARE CONTINUE HANDLER FOR NOT FOUND SET fincurs1 := 1;`
`OPEN curs1;`	

`REPEAT`
`FETCH curs1 INTO v_nbHv;`
`IF NOT fincurs1 THEN`
`SET v_tot := v_tot+v_nbHv;`
`END IF;`
`UNTIL fincurs1`
`END REPEAT;`
	`FETCH curs1 INTO v_nbHv;`
	`WHILE (NOT fincurs1) DO`
	`SET v_tot := v_tot+v_nbHv;`
	`FETCH curs1 INTO v_nbHv;`
	`END WHILE;`

`boucle1: LOOP`
`FETCH curs1 INTO v_nbHv;`
`IF (fincurs1) THEN`
`LEAVE boucle1;`
`END IF;`
`SET v_tot := v_tot+v_nbHv;`
`END LOOP;`	

`CLOSE curs1;` `SELECT v_tot AS 'Total heures pour les pilotes de ''AF''';`	

Personnellement, je préfère la programmation avec un *tant que*. L'appel de cette procédure (avec l'une ou l'autre des techniques) sur la table de la figure 7-2 produira le résultat suivant :

```
+--------------------------------------+
| Total heures pour les pilotes de 'AF' |
+--------------------------------------+
|                              3550.00 |
+--------------------------------------+
```

Accès concurrents [FOR UPDATE]

Si vous voulez verrouiller les lignes d'une table interrogée par un curseur dans le but de mettre à jour la table, sans qu'un autre utilisateur ne la modifie en même temps, il faut utiliser la clause FOR UPDATE. Elle s'emploie lors de la déclaration du curseur et verrouille les lignes concernées dès l'ouverture du curseur. Les verrous sont libérés à la fin de la transaction.

Il est souvent intéressant de pouvoir modifier facilement la ligne courante d'un curseur (UPDATE ou DELETE) à répercuter au niveau de la table. Il est conseillé d'utiliser un curseur FOR UPDATE pour verrouiller les lignes à actualiser.

Le tableau suivant décrit un bloc qui se sert du curseur FOR UPDATE pour :

- augmenter le nombre d'heures de 100 pour les pilotes de la compagnie de code 'AF' ;
- diminuer ce nombre de 100 pour les pilotes de la compagnie de code 'SING' ;
- supprimer les pilotes des autres compagnies.

Tableau 7-7 Curseur avec verrouillage explicite

Code MySQL	Commentaires

BEGIN
  DECLARE fincurs BOOLEAN DEFAULT 0;
  DECLARE v_brevet VARCHAR(6);
  DECLARE v_nbHv DECIMAL(7,2);
  DECLARE v_comp VARCHAR(4);
  DECLARE curs **CURSOR FOR**
  SELECT brevet,nbHVol,comp
 	
  FROM bdsoutou.Pilote **FOR UPDATE**;
  DECLARE CONTINUE HANDLER FOR NOT FOUND
  SET fincurs := 1;
SET AUTOCOMMIT = 0;
OPEN curs;
FETCH curs INTO v_brevet,v_nbHv,v_comp;
	Déclaration du curseur avec verrou.

WHILE (NOT fincurs) DO
  IF (v_comp='AF') THEN
  UPDATE bdsoutou.Pilote
 	
  SET nbHVol = nbHVol + 100
  WHERE brevet = v_brevet;
	Chargement et parcours du curseur.
	Mise à jour de la table Pilote par l'intermédiaire du curseur.

ELSEIF (v_comp='SING') THEN
  UPDATE bdsoutou.Pilote
  SET nbHVol = nbHVol - 100
  WHERE brevet = v_brevet;
  ELSE	

```
            DELETE FROM bdsoutou.Pilote
                  WHERE brevet = v_brevet;
   END IF;
   FETCH curs INTO v_brevet,v_nbHv,v_comp;
END WHILE;
CLOSE curs;
COMMIT;
```

Validation de la transaction.

Restrictions

- Une validation (COMMIT) avant la fermeture d'un curseur FOR UPDATE aura des effets de bord fâcheux.
- Il n'est pas possible de déclarer un curseur FOR UPDATE en utilisant dans la requête les directives DISTINCT ou GROUP BY, un opérateur ensembliste, ou une fonction d'agrégat.
- Il n'existe pas encore de directive WHERE CURRENT OF pour modifier l'enregistrement courant d'un curseur avec verrou.
- Un curseur, comme un tableau, ne peut pas être passé en paramètre d'un sous-programme ni en entrée (IN), ni en sortie (OUT).

Erreurs [codes et messages]

Avant d'aborder la gestion des exceptions, il est intéressant d'étudier le mécanisme des erreurs. Un client MySQL se voit retourner un certain nombre d'informations dès qu'il contrarie le serveur par une instruction SQL illégale. Le tableau suivant présente quelques tentatives d'extraction du nom et du nombre d'heures de vol des pilotes.

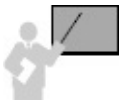
Tableau 7-8 Types d'erreurs

Instructions	Résultats
`SELECT nom,nbHVol FROM Pilote_;`	`ERROR 1146 (42S02): Table 'bdsoutou.Pilote_' doesn't exist`
`SELECT 3nom,nbHVol FROM Pilote;`	`ERROR 1054 (42S22): Unknown column '3nom' in 'field list'`
`SELECT nom,nbHVol FROME Pilote;`	`ERROR 1064 (42000): You have an error in your SQL syntax; check the manual that corresponds to your`

```
                                MySQL server version for the right syntax to use
                                near 'Pilote' at line 1
```

Le retour (contrarié) du serveur message contient trois informations :

- Un code numérique signalé par le titre `ERROR` (dans les exemples du tableau précédent, 1146, 1054, etc.). Vous trouverez dans l'index de cet ouvrage une entrée regroupant quelques exemples pour les codes d'erreurs usuels. Attention, ce nombre est propre à MySQL, et donc ne représente rien au regard d'un autre SGBD.
- Une chaîne de 5 caractères signalée par le titre `SQLSTATE` (dans les exemples du tableau précédent, '42S02', '42S22' et '42000'). Ces valeurs sont compatibles avec la norme ANSI SQL et la spécification ODBC.
- Une chaîne de caractères fournissant la description textuelle de l'erreur.



Tous les codes d'erreurs MySQL n'ont pas un code `SQLSTATE` associé. La valeur du code `SQLSTATE` est alors `'HY000'` (définissant une erreur générale). On y trouvera notamment toutes les erreurs relatives au système de fichiers, à la mémoire, au processus, etc.

MySQL utilise pour l'heure des numéros d'erreurs allant de 1 000 à 2 999. En conséquence, évitez cette plage pour définir vos propres exceptions par des entiers positifs (voir le paragraphe consacré à `SIGNAL`).

Par ailleurs, le code `SQLSTATE` représente en fait une famille d'erreurs si bien que plusieurs erreurs distinctes auront le même `SQLSTATE`. On est donc bien loin d'une portabilité aisée, même pour les SGBD qui annoncent une compatibilité SQL ANSI…

Vous trouverez dans l'annexe B (*Error Codes, and Common Problems*) de la documentation la liste des erreurs avec leur libellé. Seuls quelques scénarios classiques sont résolus.

Exceptions

Afin d'éviter qu'un programme ne s'arrête dès la première erreur suite à une

instruction SQL (SELECT ne retournant aucune ligne, INSERT ou UPDATE d'une valeur incorrecte, DELETE d'un enregistrement « père » ayant des enregistrements « fils » associés, etc.), il est indispensable de prévoir les cas potentiels d'erreurs et d'associer à chacun de ces cas la programmation d'une exception (*handler* dans le vocabulaire de MySQL).

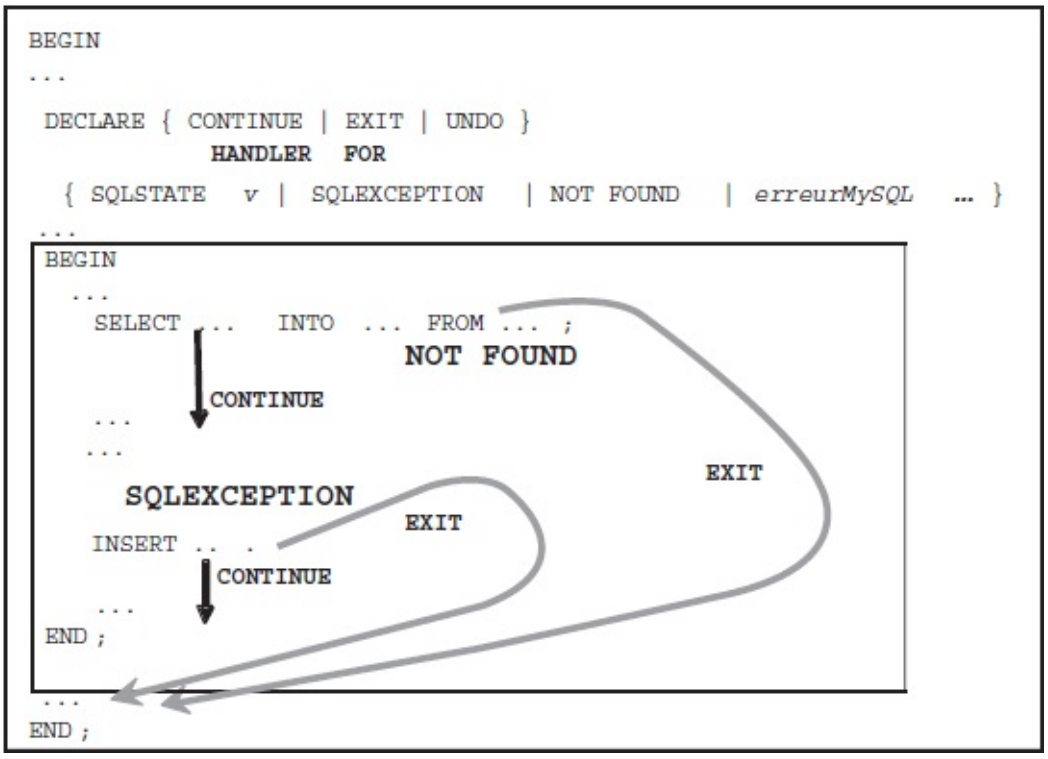
Dans le langage des informaticiens, on dit qu'on *garde la main* pendant l'exécution du programme. Le mécanisme des exceptions (*handling errors*) est largement utilisé par tous les développeurs, car il est prépondérant dans la mise en œuvre des transactions. Les exceptions peuvent se paramétrer dans un sous-programme (fonction ou procédure cataloguée) ou un déclencheur.

Généralités

Une exception MySQL correspond à une condition d'erreur et peut être associée à un identificateur (exception nommée). Une exception est détectée (aussi dite « levée ») si elle est prévue dans un *handler* au cours de l'exécution d'un bloc (entre BEGIN et END). Une fois levée, elle fait continuer (ou sortir du bloc) le programme après avoir réalisé une ou plusieurs instructions que le programmeur aura explicitement spécifiées.

La figure suivante illustre les deux mécanismes qui peuvent être mis en œuvre pour gérer une exception (seuls CONTINUE et EXIT sont actuellement pris en charge par MySQL). Supposons que l'extraction ne ramène aucune ligne, on peut programmer la sortie du bloc courant ou continuer dans le bloc. Supposons que l'insertion déclenche une erreur, on peut également décider de sortir ou de poursuivre le traitement.

***Figure 7-4** Principe général des exceptions*



Si aucune erreur ne se produit, le traitement se termine ou retourne à son appelant, s'il s'agit d'un sous-programme lancé d'un programme principal. La syntaxe générale d'une exception est la suivante. Les exceptions doivent être déclarées de préférence après les variables et avant les curseurs.

```
DECLARE { CONTINUE | EXIT | UNDO }
        HANDLER FOR
    { SQLSTATE [VALUE] 'valeur_sqlstate' | nomException | SQLWARNING
      | NOT FOUND | SQLEXCEPTION | code_erreur_mysql }
   instructions MySQL;
   [, { SQLSTATE...} ...]
```

- La directive CONTINUE (appelée *handler*) force à poursuivre l'exécution de programme lorsqu'il se passe un événement prévu dans la clause FOR.
- Le *handler* EXIT fait sortir l'exécution du bloc courant (entre BEGIN et END).

Le *handler* UNDO n'est pas encore reconnu. À son nom, on se doute de son utilité, à savoir défaire les instructions SQL qui auront été exécutées (sans avoir été validées par un COMMIT) avant que l'exception ne se déclenche.

- SQLSTATE permet de couvrir toutes les erreurs d'un état donné.
- *nomException* s'applique à la gestion des exceptions nommées (étudiées plus loin).
- SQLWARNING permet de couvrir toutes les erreurs d'état SQLSTATE débutant par 01.
- NOT FOUND permet de couvrir toutes les erreurs d'état SQLSTATE débutant par 02 (relatives à la gestion des curseurs).
- SQLEXCEPTION gère toutes les erreurs qui ne sont ni gérées par SQLWARNING ni par NOT FOUND.
- instructions MySQL : une ou plusieurs instructions du langage de MySQL (bloc, appel possibles par CALL d'une fonction ou d'une procédure cataloguée).

Il est possible de grouper plusieurs déclarations d'exceptions, ainsi que de prévoir plusieurs conditions pour une même exception. En plus de pouvoir tester des erreurs pour tel ou tel état (SQLSTATE), nous verrons que l'on peut récupérer une erreur de code donné (paramètre code_erreur_mysql). Par exemple, ERROR 1046 désigne la non sélection d'une base de données (1046 devra être écrit en lieu et place de code_erreur_mysql).

Étudions à présent les différents types d'exceptions en programmant des procédures simples interrogeant la table Pilote illustrée à la figure 7-2.

Exceptions avec EXIT

Examinons la clause EXIT de l'instruction DECLARE HANDLER à travers un exemple.

Gestion de NOT FOUND

Le tableau suivant décrit une procédure qui gère une erreur : aucun pilote n'est associé à la compagnie de code passé en paramètre (NOT FOUND). La procédure ne se termine pas correctement si plusieurs lignes sont retournées (erreur Result consisted of more than one row).

Tableau 7-9 Exception NOT FOUND traitée avec EXIT

Code MySQL	Commentaires

`CREATE PROCEDURE bdsoutou.procException1`
`(IN p_comp VARCHAR(4))`
`BEGIN`
`DECLARE flagNOTFOUND BOOLEAN DEFAULT 0;`
`DECLARE var1 VARCHAR(20);`
	Déclaration de la procédure et des variables.

`BEGIN`
`DECLARE EXIT HANDLER FOR NOT FOUND`
`SET flagNOTFOUND :=1;`
	Bloc qui déclare l'exception.

`SELECT nom INTO var1 FROM bdsoutou.Pilote` `WHERE comp=p_comp;`	Requête pouvant déclencher l'exception prévue.

`SELECT CONCAT('Le seul pilote de la compagnie ',`
`p_comp,' est ',var1)`
`AS 'Resultat procException1';`
	Affichage du résultat.

`END;`	Fin du bloc.

`IF flagNOTFOUND THEN`
`SELECT CONCAT('Il n''y a pas de pilote pour la`
`compagnie ',p_comp)`
`AS 'Resultat procException1';`
	Gestion de l'erreur.

`END IF;` `END;`	Fin de la procédure.

La trace d'une exécution correcte de cette procédure (si la requête retourne une seule ligne, la procédure ne passe pas dans le *si*) est la suivante :

```
CALL bdsoutou.procException1('CAST')$
+-----------------------------------------------+
| Resultat procException1                       |
+-----------------------------------------------+
| Le seul pilote de la compagnie CAST est Labat |
+-----------------------------------------------+
```

Une autre exécution correcte de cette procédure (qui se déroute hors du bloc du fait de `NOT FOUND`) est la suivante :

```
CALL bdsoutou.procException1('RIEN')$
+-----------------------------------------------+
| Resultat procException1                       |
+-----------------------------------------------+
| Il n'y a pas de pilote pour la compagnie RIEN |
+-----------------------------------------------+
```

Gestion d'une erreur particulière

Le tableau suivant décrit une amélioration de la précédente procédure par le fait de gérer l'erreur particulière permettant de savoir si la requête renvoie plusieurs lignes (`ERROR 1172(42000): Result consisted of more than one row`). La

procédure se termine maintenant correctement si la requête retourne une seule ligne ou plusieurs (message personnalisé en sortie de bloc).

Tableau 7-10 Exceptions 1172 et `NOT FOUND` traitées avec `EXIT`

Code MySQL	Commentaires

`CREATE PROCEDURE bdsoutou.procException1`
`(IN p_comp VARCHAR(4))`
`BEGIN`
`DECLARE flagPlusDun BOOLEAN DEFAULT 0;`
`DECLARE flagNOTFOUND BOOLEAN DEFAULT 0;`
`DECLARE var1 VARCHAR(20);`
	Déclaration de la procédure et des variables.

`BEGIN`
`DECLARE EXIT HANDLER FOR 1172`
`SET flagPlusDun :=1;`
`DECLARE EXIT HANDLER FOR NOT FOUND`
`SET flagNOTFOUND :=1;`
	Bloc qui déclare les deux exceptions.

`SELECT nom INTO var1 FROM bdsoutou.Pilote` `WHERE comp=p_comp;`	Requête pouvant déclencher l'exception prévue.

`SELECT CONCAT('Le seul pilote de la compagnie ',`
`p_comp,' est ',var1)`
`AS 'Resultat procException1';`
	Affichage du résultat.

`END;`	Fin du bloc.

`IF flagNOTFOUND THEN`
`SELECT CONCAT('Il n''y a pas de pilote pour la`
`compagnie ',p_comp)`
`AS 'Resultat procException1';`
`END IF;`
	Gestion des erreurs.

`IF flagPlusDun THEN`
`SELECT CONCAT('Il y a plusieurs pilotes pour la`
`compagnie ',p_comp)`
`AS 'Resultat procException1';`
`END IF;`	

`END;`	Fin de la procédure.

La trace d'une exécution correcte de cette procédure (qui se déroute hors du bloc du fait de plusieurs lignes retournées) est la suivante :

```
CALL bdsoutou.procException1('AF')$
+-----------------------------------------------+
| Resultat procException1                       |
+-----------------------------------------------+
| Il y a plusieurs pilotes pour la compagnie AF |
+-----------------------------------------------+
```

Appel d'une procédure dans l'exception

Le tableau suivant décrit l'appel d'une procédure dans le cas de l'erreur NOT FOUND. La procédure principale exécute le sous–programme, puis sort du bloc principal et se termine correctement.

Tableau 7-11 Exception NOT FOUND traitée avec EXIT et CALL

Code MySQL	Commentaires

`CREATE PROCEDURE bdsoutou.pasTrouve`
`(IN p_comp VARCHAR(4))`
`BEGIN`
`SELECT CONCAT('Il n''y a pas de pilote pour`
`la compagnie ',p_comp)`
`AS 'Resultat procpasTrouve';`
`END;`
	Codage du sous-programme appelé lors de l'exception.

`CREATE PROCEDURE bdsoutou.procException1`
`(IN p_comp VARCHAR(4))`
`BEGIN`
`DECLARE var1 VARCHAR(20);`
`DECLARE EXIT HANDLER FOR NOT FOUND`
`CALL bdsoutou.pasTrouve(p_comp);`
	Procédure qui déclare l'exception.

`SELECT nom INTO var1 FROM bdsoutou.Pilote` `WHERE comp = p_comp;`	Requête pouvant déclencher l'exception prévue.

`SELECT CONCAT('Le seul pilote de la compagnie`
`',p_comp,' est ',var1)`
`AS 'Resultat procException1';`
	Affichage du résultat.

`END;`	Fin de la procédure principale.

La trace d'une exécution correcte de cette procédure (qui se déroute vers le sous-programme appelé du fait de NOT FOUND) est la suivante :

```
+-------------------------------------------+
| Resultat procpasTrouve                    |
+-------------------------------------------+
| Il n'y a pas de pilote pour la compagnie RIEN |
+-------------------------------------------+
```

Il est aisé de transposer ce raisonnement à plusieurs exceptions appelant différents sous-programmes.

Exceptions avec CONTINUE

Étudions à présent la clause CONTINUE de l'instruction DECLARE HANDLER.

Gestion de NOT FOUND

Le tableau suivant décrit la même procédure gérant l'erreur NOT FOUND. La procédure se termine correctement si la requête retourne une seule ligne, mais pas lorsqu'elle en renvoie plusieurs (erreur Result consisted of more than one row).

Tableau 7-12 Exception NOT FOUND traitée avec CONTINUE

Code MySQL	Commentaires

`CREATE PROCEDURE bdsoutou.procException1`
`(IN p_comp VARCHAR(4))`
`BEGIN`
`DECLARE flagNOTFOUND BOOLEAN DEFAULT 0;`
`DECLARE var1 VARCHAR(20);`
	Déclaration de la procédure et des variables.

`DECLARE CONTINUE HANDLER FOR NOT FOUND` `SET flagNOTFOUND :=1;`	Bloc qui déclare l'exception.
`SELECT nom INTO var1 FROM bdsoutou.Pilote` `WHERE comp=p_comp;`	Requête pouvant déclencher l'exception prévue.

`IF flagNOTFOUND THEN`
`SELECT CONCAT('Il n''y a pas de pilote pour la`
`compagnie ',p_comp)`
`AS 'Resultat procException1';`
`ELSE`
`SELECT CONCAT('Le seul pilote de la compagnie ',`
`p_comp,' est ',var1)`
`AS 'Resultat procException1';`
`END IF;`
	Test de gestion de l'erreur.
	Affichage du résultat.

`END;`	Fin de la procédure.

Gestion d'une erreur particulière

Le tableau suivant décrit la même procédure qui gère en plus l'erreur Result consisted of more than one row. La procédure se termine correctement si la requête retourne une seule ou plusieurs lignes.

Tableau 7-13 Exceptions 1172 et NOT FOUND traitées avec CONTINUE

Code MySQL	Commentaires

`CREATE PROCEDURE bdsoutou.procException1`
`(IN p_comp VARCHAR(4))`
`BEGIN`
`DECLARE flagNOTFOUND BOOLEAN DEFAULT 0;`
`DECLARE flagPlusDun BOOLEAN DEFAULT 0;`
`DECLARE var1 VARCHAR(20);`
	Déclaration de la procédure et des variables.

```
  DECLARE CONTINUE HANDLER FOR 1172
        SET flagPlusDun :=1;


DECLARE CONTINUE HANDLER FOR NOT FOUND
          SET flagNOTFOUND :=1;


SELECT nom INTO var1 FROM bdsoutou.Pilote
          WHERE comp=p_comp;


IF flagNOTFOUND THEN
   SELECT CONCAT('Il n''y a pas de pilote pour la
          compagnie ',p_comp)
          AS 'Resultat procException1';
ELSEIF flagPlusDun THEN
       SELECT CONCAT('Il y a plusieurs pilotes
              pour la compagnie ',p_comp)
              AS 'Resultat procException1';
  ELSE
       SELECT CONCAT('Le seul pilote de la compagnie
                    ',p_comp,' est ',var1)
                    AS 'Resultat procException1';
END IF;

END;
```

Bloc qui déclare les deux exceptions.

Requête pouvant déclencher l'exception prévue.

Test de gestion des erreurs.

Affichage du résultat.

Fin de la procédure.

Gestion des autres erreurs [SQLEXCEPTION]

Si une erreur non prévue en tant qu'exception (dans les clauses DECLARE HANDLER) se produisait, le programme se terminerait anormalement en renvoyant l'erreur en question. La directive SQLEXCEPTION couvre toutes les erreurs qui ne sont administrées ni par SQLWARNING ni par NOT FOUND.

Dans notre exemple (sélection d'une colonne d'une table), seule une erreur « interne » pourrait éventuellement se produire (base de données ou table inexistante, colonne de la table inexistante ou d'un type différent de VARCHAR(4) ou autre erreur système).

MySQL ne permet pas, pour l'instant, de récupérer dynamiquement, au sein d'un sous-programme, le code et le message de l'erreur associée à une exception levée suite à une instruction SQL, et qui n'a pas été prévue dans un

handler.

En d’autres termes, il faudra soit :

- interrompre brusquement votre procédure avec différents cas d’erreurs, pour lister les codes erreur générés en sortie ;
- vous contenter de gérer globalement les autres exceptions, tout en ne sachant pas de quelles erreurs il s’agit.

Le tableau suivant décrit la précédente procédure qui contrôle en plus toutes les autres erreurs non prévues, en appelant le sous-programme `autreErreur()`.

Tableau 7-14 Exceptions toutes traitées avec `EXIT` et `SQLEXCEPTION`

Code MySQL	Commentaires

`CREATE PROCEDURE bdsoutou.tropdeLignes`
`(IN p_comp VARCHAR(4))`
`BEGIN`
`SELECT CONCAT('Il y a plusieurs pilotes pour`
`la compagnie ',p_comp)`
`AS 'Resultat tropdeLignes';`
`END;`
	Codage des sous-programmes appelés lors des exceptions.

`CREATE PROCEDURE bdsoutou.pasTrouve`
`(IN p_comp VARCHAR(4))`
`BEGIN`
`SELECT CONCAT('Il n''y a pas de pilote pour`
`la compagnie ',p_comp)`
`AS 'Resultat pasTrouve';`
`END;`	

`CREATE PROCEDURE bdsoutou.autreErreur()`
`BEGIN`
`SELECT 'Erreur mais laquelle?'`
`AS 'Resultat autreErreur';`
`END;`
	3 cas

`CREATE PROCEDURE bdsoutou.procException1`
`(IN p_comp VARCHAR(4))`
`BEGIN`
`DECLARE var1 VARCHAR(20);`
`DECLARE EXIT HANDLER FOR 1172`
`CALL bdsoutou.tropdeLignes(p_comp);`
`DECLARE EXIT HANDLER FOR NOT FOUND`
`CALL bdsoutou.pasTrouve(p_comp);`
`DECLARE EXIT HANDLER FOR SQLEXCEPTION`
`CALL bdsoutou.autreErreur();`
`SELECT nom INTO var1 FROM bdsoutou.Pilote`
`WHERE comp = p_comp;`
`SELECT CONCAT('Le seul pilote de la compagnie',`
`p_comp,' est ',var1)`
`AS 'Resultat procException1';`
	Procédure principale qui déclare les deux exceptions et toutes les autres.
	Requête pouvant déclencher l'exception prévue.
	Affichage du résultat.

`END;`	Fin de la procédure.

La trace d'une exécution de cette procédure (quelle que soit la valeur du paramètre passé en entrée), suite à la suppression de la table `Pilote` dans la base `bdsoutou`, est la suivante :

```
CALL bdsoutou.procException1('AF')$
+-----------------------+
| Resultat autreErreur  |
+-----------------------+
| Erreur mais laquelle? |
+-----------------------+
```

Même erreur sur différentes instructions

Plusieurs cas de figure sont possibles suivant qu'on désire maîtriser une exception ou terminer toutes les exceptions avant la fin du sous-programme.

Gestion d'une seule exception

Le tableau suivant décrit une procédure qui gère deux fois l'erreur non trouvée (NOT FOUND) sur deux requêtes distinctes. La première requête extrait le nom du pilote de code passé en paramètre. La deuxième donne le nom du pilote ayant un nombre d'heures de vol égal à celui passé en paramètre. Le sous-programme se termine correctement si les deux requêtes ne retournent qu'un seul enregistrement. Les autres erreurs potentielles ne sont pas prises en compte.

Le principe est d'utiliser une variable indiquant quelle est la requête qui a fait sortir du bloc par l'exception levée.

Exécutons cette procédure avec différents paramètres, on obtient :

```
CALL bdsoutou.procException2('PL-1', 1000)$
+-------------------------------------------------------------------+
| CONCAT('Le pilote de  code ',p_brevet,' est ',v_nom)              |
+-------------------------------------------------------------------+
| Le pilote de  code PL-1 est Roux                                  |
+-------------------------------------------------------------------+
+-------------------------------------------------------------------+
| CONCAT('Le pilote ayant ',p_heures,' heures de vol est ',v_nom)   |
+-------------------------------------------------------------------+
| Le pilote ayant 1000 heures de vol est Mansalier                  |
+-------------------------------------------------------------------+
CALL bdsoutou.procException2('PL-0', 2450)$
+-------------------------------------------------------------------+
| CONCAT('Pas de pilote de brevet : ',p_brevet)                     |
+-------------------------------------------------------------------+
| Pas de pilote de brevet : PL-0                                    |
+-------------------------------------------------------------------+
```

Tableau 7-15 Une exception NOT FOUND traitée pour deux instructions

Code MySQL	Commentaires

`CREATE PROCEDURE bdsoutou.procException2`
`(IN p_brevet VARCHAR(6),IN p_heures DECIMAL(7,2))`
`BEGIN`
`DECLARE v_nom VARCHAR(20);`
`DECLARE flagNOTFOUND BOOLEAN DEFAULT 0;`
`DECLARE v_requete TINYINT;`	

`BEGIN`
`DECLARE EXIT HANDLER FOR NOT FOUND SET flagNOTFOUND :=1;`
`SET v_requete := 1;`	

```
  SELECT nom INTO v_nom FROM bdsoutou.Pilote
            WHERE brevet = p_brevet;
  SELECT CONCAT('Le pilote de  code ',p_brevet,
                 ' est ',v_nom);
  SET v_requete := 2;
  SELECT nom INTO v_nom FROM bdsoutou.Pilote
            WHERE nbHVol = p_heures;
  SELECT CONCAT('Le pilote ayant ',p_heures,
                ' heures de vol est ',v_nom);
END;
```

Bloc avec les requêtes déclenchant potentiellement une exception prévue.

```
IF flagNOTFOUND THEN
   IF v_requete = 1 THEN
    SELECT CONCAT('Pas de pilote de brevet : ',p_brevet);
   ELSEIF v_requete = 2 THEN
    SELECT CONCAT('Pas de pilote ayant ce nombre
           d''heures de vol : ',p_heures);
   END IF;
 END IF;
END;
```

Traitement pour savoir quelle requête a provoqué l'exception.

Dans cette procédure, une erreur sur la première requête interrompt le programme (après avoir traité l'exception), et de ce fait la deuxième requête n'est pas évaluée. Pour cela, il est intéressant d'utiliser des blocs imbriqués pour poursuivre le traitement après avoir traité une ou plusieurs exceptions.

Gestion de plusieurs exceptions

Le tableau suivant décrit une procédure qui traite toutes les mêmes exceptions en séquence. Ce mécanisme permet de continuer l'exécution après que MySQL a levé une exception. Dans cette procédure, les deux requêtes sont évaluées indépendamment du résultat retourné par chacune d'elles.

Tableau 7-16 Exceptions `NOT FOUND` toutes traitées

Code MySQL	Commentaires

`CREATE PROCEDURE bdsoutou.procException3`
`      (IN p_brevet VARCHAR(6),IN p_heures DECIMAL(7,2))`
`BEGIN`
`  DECLARE v_nom        VARCHAR(20);`
`  DECLARE flagNOTFOUND BOOLEAN DEFAULT 0;`
`  DECLARE **CONTINUE** HANDLER FOR **NOT FOUND**`
`         SET flagNOTFOUND := 1;`
`  SELECT nom INTO v_nom FROM bdsoutou.Pilote`
`            WHERE brevet = p_brevet;`
`  IF flagNOTFOUND THEN`
`     SELECT CONCAT('Pas de pilote de brevet : ',p_brevet);`
`     SET flagNOTFOUND := 0;`
`  ELSE`
`     SELECT CONCAT('Le pilote de  code ',p_brevet,`
`                   ' est ',v_nom);`
`  END IF;`
	Gestion de l'exception de la première requête.

```
   SELECT nm INTO v_nom FROM bdsoutou.Pilote
             WHERE nbHVol = p_heures;
   IF flagNOTFOUND THEN

      SELECT CONCAT('Pas de pilote ayant ce nombre d'
                    'heures de vol : ',p_heures);
   ELSE
      SELECT CONCAT('Le pilote ayant ',p_heures,
                    ' heures de vol est ',v_nom);
   END IF;
 END;
```

Gestion de l'exception de la deuxième requête.

L'exécution suivante de cette procédure déclenche les deux exceptions (ce qui n'était pas le cas dans la procédure précédente).

```
CALL bdsoutou.procException3('PL-0', 500)$
+-------------------------------------------+
| CONCAT('Pas de pilote de brevet : ',p_brevet) |
+-------------------------------------------+
| Pas de pilote de brevet : PL-0                |
+-------------------------------------------+
+-------------------------------------------------------------------+
| CONCAT('Pas de pilote ayant ce nombre d''heures de vol : ',p_heures)  |
+-------------------------------------------------------------------+
| Pas de pilote ayant ce nombre d'heures de vol : 500              |
+-------------------------------------------------------------------+
```

Il resterait à programmer l'exception `ERROR 1172 (Result consisted of more than one row`) pour gérer, au niveau de la seconde requête, l'extraction de plusieurs pilotes ayant un même nombre d'heures de vol. Deux solutions s'offrent à vous : dans un sous-bloc avec `EXIT` ou dans le même bloc avec un `CONTINUE`, et une variable pour tester l'éventuelle redirection.

Afin de déclarer d'autres exceptions, il faut consulter la liste des erreurs dans la documentation officielle (*Appendix B. Error Codes and Messages*) de manière à connaître le numéro d'erreur MySQL (paramètre `code_erreur_mysql` dans la syntaxe `DECLARE HANDLER`).

Vous pouvez aussi écrire un bloc qui programme volontairement l'erreur pour voir, sous l'interface de commande, le numéro que MySQL renvoie.

Exceptions nommées

Pour intercepter une erreur MySQL et lui attribuer au passage un identificateur,

il faut utiliser la clause `DECLARE CONDITION`. La syntaxe est la suivante :

Déclaration

```
DECLARE nomException CONDITION FOR
        {SQLSTATE [VALUE] 'valeur_sqlstate' | code_erreur_mysql}
```

Il est ainsi possible de regrouper plusieurs types d'erreurs (avec `SQLSTATE` ou cibler une erreur en particulier en indiquant le code erreur de MySQL). Une fois l'exception nommée, il est possible de l'utiliser dans la déclaration de l'événement associé via la directive `DECLARE HANDLER`.

Déclenchement

Considérons les deux tables suivantes. La colonne `comp` de la table `Pilote` est une clé étrangère vers la table `Compagnie`. Programmons une procédure qui supprime une compagnie de code passé en paramètre.

***Figure 7-5** Deux tables*

Compagnie

comp	ville	nomComp
AF	Paris	Air France
SING	Singapour	Singapore AL
CAST	Blagnac	Castanet AL
EJET	**Dublin**	**Easy Jet**

à détruire

Pilote

brevet	nom	nbHVol	*comp*
PL-1	Roux	2450	AF
PL-2	Peyrard	900	AF
PL-3	Mansalier	1000	SING
PL-4	Labat	2450	CAST
PL-5	Marti	200	AF
PL-6	Brulin	2450	SING

Le tableau suivant décrit la procédure `procExceptionNommee` qui intercepte une erreur référentielle (`SQLSTATE` à 23 000). Il s'agit de contrôler le programme si la compagnie à détruire est encore rattachée à un enregistrement référencé dans la table `Pilote`.

Tableau 7-17 Programmation d'une exception nommée

Code MySQL	Commentaires

`CREATE PROCEDURE bdsoutou.procExceptionNommee`
`(IN p_comp VARCHAR(4))`
`BEGIN`
`DECLARE flagerr BOOLEAN DEFAULT 0;`
`BEGIN`
`DECLARE erreur_ilResteUnPilote CONDITION`
`FOR SQLSTATE '23000';`
`DECLARE EXIT HANDLER`
`FOR erreur_ilResteUnPilote SET flagerr :=1;`
`SET AUTOCOMMIT=0;`
`DELETE FROM Compagnie WHERE comp = p_comp;`
`SELECT CONCAT('Compagnie ',p_comp, ' détruite')`
`AS 'Resultat procExceptionNommee';`
`END;`
`IF flagerr THEN`
`ROLLBACK;`
`SELECT CONCAT('Désolé, il reste encore un pilote à`
`la compagnie ',p_comp)`
`AS 'Resultat procExceptionNommee';`
`ELSE`
`COMMIT;`
`END IF;`
`END;`
	Déclaration de l'exception nommée.
	
	Corps du traitement (validation).
	
	Gestion de l'exception.

Les traces de l'exécution de cette procédure sont les suivantes. Notez que si on appelle cette procédure en passant en paramètre une compagnie inexistante, le sous-programme se termine normalement sans passer dans la section d'erreur.

```
CALL bdsoutou.procExceptionNommee('AF')$
+-----------------------------------------------------+
| Resultat procExceptionNommee                        |
+-----------------------------------------------------+
| Désolé, il reste encore un pilote à la compagnie AF |
+-----------------------------------------------------+
CALL bdsoutou.procExceptionNommee('EJET')$
+-----------------------------+
| Resultat procExceptionNommee |
+-----------------------------+
| Compagnie EJET détruite      |
+-----------------------------+
```

Déroutements [SIGNAL et RESIGNAL]

Dérouter avec SIGNAL

Disponible depuis la version 5.5, la directive `SIGNAL` fournit un moyen de retourner (après sortie du bloc d'instructions) une erreur applicative (numéro

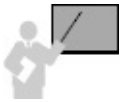
d'erreur, code d'état et texte du message) depuis un sous-programme (et également dans un déclencheur) suite à une exception (*handler*).

Oracle implémente ce concept avec RAISE, SQL Server utilise RAISERROR et IBM suit la norme SQL (comme MySQL) sur ce point en disposant aussi de SIGNAL. La syntaxe de cette directive est la suivante :

```
SIGNAL SQLSTATE [VALUE] 'v_sqlstate' | nom_exception
       [SET { MESSAGE_TEXT | MYSQL_ERRNO } = valeur
            [, …] ]
```

La valeur v_sqlstate à retourner est soit codifiée sur 5 caractères (code SQLSTATE), soit elle prend la forme d'un nom d'exception. Vous devez respecter les conventions suivantes relatives aux deux premiers caractères de la valeur du code SQLSTATE :

- '00' indique un traitement correct (*success*).
- '01' indique un avertissement (*warning*) qui ne sera lisible qu'avec un SHOW WARNINGS.
- '02' indique un traitement de type non trouvé (*not found*), valable uniquement dans un contexte de curseurs.
- Autre pour indiquer une exception.



Lorsque vous utilisez un nom d'exception, il faut que cette exception soit au préalable avec l'option SQLSTATE (utilisez le code 45000) et non pas avec l'option d'un numéro d'erreur.

La procédure suivante déroute une erreur (avec le numéro 9001) et un *warning* (avec le numéro 9000). L'erreur s'adresse aux pilotes qui totalisent moins de 1 000 heures de vol (exemple du PL-5). L'avertissement concerne les pilotes qui n'appartiennent pas à la compagnie Air France (exemple du PL-6). Les pilotes d'Air France totalisant plus de 1 000 heures de vol ne donnent pas lieu à un déroutement (exemple du PL-1).

Outre un code d'état, un numéro d'erreur et un texte décrivant l'erreur, d'autres valeurs peuvent être retournées par d'autres paramètres de retour. Citons notamment SCHEMA_NAME (un nom de base), CONSTRAINT_NAME (un nom de contrainte), TABLE_NAME (un nom de table), COLUMN_NAME (un nom de colonne) et

CURSOR_NAME (un nom de curseur). Ainsi, la première instruction SIGNAL de l'exemple précédent pourrait être complétée de la manière suivante :

```
SIGNAL pilote_trop_bleu
            SET MESSAGE_TEXT='Pilote trop peu experimente.',
                MYSQL_ERRNO = 1001,
                SCHEMA_NAME='bdsoutou',
                TABLE_NAME='Pilote',
                COLUMN_NAME='nbHVol';
```

Tableau 7-18 Déroutements

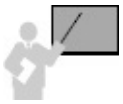
Procédure	Appels

```
CREATE PROCEDURE exsignal1
       (IN p_brevet VARCHAR(6))
BEGIN
  DECLARE v_nbHv   DECIMAL(7,2);
  DECLARE v_comp   VARCHAR(4);
  DECLARE pilote_trop_bleu
         CONDITION FOR SQLSTATE '45000';
  SELECT comp,nbHVol INTO v_comp,v_nbHv
       FROM Pilote WHERE brevet =
p_brevet;
  IF (v_nbHv < 1000) THEN
     SIGNAL pilote_trop_bleu
        SET MESSAGE_TEXT='Pilote trop peu
experimente.',
            MYSQL_ERRNO = 9001;
   ELSEIF (v_comp != 'AF') THEN
     SIGNAL SQLSTATE '01000'
        SET MESSAGE_TEXT = 'Warning, pas
Air France',
            MYSQL_ERRNO = 9000;
    ELSE
     SELECT p_brevet,v_nbHv,v_comp;
   END IF;
END;
$
```

```
mysql> CALL exsignal1('PL-5')$
ERROR 9001 (45000): Pilote trop peu
experimente.
mysql> CALL exsignal1('PL-6')$
Query OK, 0 rows affected, 1 warning
mysql> SHOW WARNINGS$
+---------+------+-----------------+
| Level   | Code | Message         |
+---------+------+-----------------+
| Warning | 9000 | Warning, pas Air
                    France          |
+---------+------+-----------------+
mysql> CALL exsignal1('PL-1')$
+----------+---------+--------+
| p_brevet | v_nbHv  | v_comp |
+----------+---------+--------+
| PL-1     | 2550.00 | AF     |
+----------+---------+--------+
```

Pour l'instant, la documentation ne précise rien à propos de la manière d'accéder à ces remontées d'informations. La future procédure GET DIAGNOSTICS (nom issu de la norme SQL) devrait répondre à ce besoin.

Étudions à présent le moyen de récupérer la « main » (et non le *main*) après l'appel d'un sous-programme renvoyant un signal.



Afin de capturer le retour d'un signal provenant d'un sous-programme appelé, vous devez déclarer un *handler* équivalent en termes de `SQLSTATE` dans le sous-programme appelant.

La procédure précédente est appelée dans la procédure suivante. Trois retours sont possibles : une erreur (transmise par `SIGNAL pilote_trop_bleu…`), un *warning* (transmis par `SIGNAL SQLSTATE '01000'…`) et un code de retour correct (fin du sous-programme sans émission d'un signal). Deux *handlers* seront donc nécessaires dans le sous-programme appelant.

Tableau 7-19 Récupération d'un signal

Procédure	Appels
`CREATE PROCEDURE exsignal2` `(IN p_brevet VARCHAR(6))` `BEGIN` `DECLARE v_flag TINYINT DEFAULT 0;` `DECLARE CONTINUE HANDLER` `FOR SQLSTATE '01000' SET v_flag = 1;` `DECLARE CONTINUE HANDLER` `FOR SQLSTATE VALUE '45000' SET v_flag = 2;` `CALL exsignal1(p_brevet);` `CASE` `WHEN v_flag=0 THEN SET v_flag = 0; /* Rien à faire */` `WHEN v_flag=1 THEN SHOW WARNINGS;` `WHEN v_flag=2 THEN SHOW ERRORS;` `END CASE;` `END;`	`mysql > CALL exsignal2('PL-5')$` `+-------+------+-----------------+` `| Level	Code	Message |` `+-------+------+-----------------+` `| Error	9001	Pilote trop peu experimente. |` `mysql > CALL exsignal2('PL-6')$` `+---------+------+----------------+` `| Level	Code	Message |` `+---------+------+----------------+` `| Warning	9000	Warning, pas Air France |` `mysql > CALL exsignal2('PL-1')$` `+----------+---------+--------+` `| p_brevet	v_nbHv	v_comp |` `+----------+---------+--------+` `| PL-1	2450.00	AF |`

Notez qu'il est aussi possible d'utiliser `FOR SQLWARNING` à la place de `FOR SQLSTATE '01000'`.

Afin de capturer le retour d'un signal provenant d'un sous-programme appelé, vous devez déclarer un *handler* équivalent en termes de `SQLSTATE` dans le sous-

programme appelant.

Si vous appelez une telle procédure dans un programme Java (voir le chapitre 8), vous pourrez utiliser la méthode `getErrorCode()`. Dans le cas d'un programme PHP (voir chapitre 9), la fonction `mysqli_errno()` vous permettra de traiter un signal retourné si un numéro d'erreur lui est affecté.

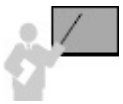
Rerouter avec RESIGNAL

Disponible en même temps que `SIGNAL` (version 5.5), la directive `RESIGNAL` fournit un moyen de rerouter un signal déjà émis. Ce mécanisme est très utile dès que vous désirez associer un bloc d'instructions à une exception, par exemple, ou dès que vous appelez un sous-programme qui renvoie une erreur via `SIGNAL`. Dans la syntaxe de `RESIGNAL`, les paramètres sont identiques à `SIGNAL`.

```
RESIGNAL [SQLSTATE [VALUE] 'valeur_sqlstate' | nom_exception
        [SET { MESSAGE_TEXT | MYSQL_ERRNO } = valeur
           [, …] ] ]
```

Trois cas sont possibles :

- Sans paramètre : le signal en cours est rerouté à l'appelant sans en changer le contenu (numéro d'erreur, code d'état, texte du message, nom de la table ,etc.).
- Avec des paramètres d'un nouveau signal sans changer le code d'état (`SQLSTATE`) (nouveau numéro d'erreur, texte du message, nom de la table, etc.) : le signal en cours est transformé par les caractéristiques du nouveau avant d'être rerouté à l'appelant.
- Avec un nouveau code d'état (`SQLSTATE`) : le signal en cours est transformé par les caractéristiques du nouveau avant d'être rerouté à l'appelant.



Ces trois formes d'utilisation de `RESIGNAL` nécessitent qu'un *handler* soit actif pendant l'exécution. Dans le cas contraire, l'erreur suivante est retournée : `ERROR 1645 (0K000): RESIGNAL when handler not active.`

Le fonctionnement des déroutements est basé sur une pile (stockant une zone appelée *diagnostic area*). Dès qu'un *handler* prend le contrôle (une exception

est levée), il empile le nouveau diagnostic. Le nombre maximal de conditions dans une zone *diagnostic area* est determiné par la variable système `max_error_count` (par défaut 64).

Avant de traiter des exemples pour illustrer chacun de ces trois cas, intéressons-nous au fonctionnement général de `RESIGNAL`.

RESIGNAL pour les blocs associés à des exceptions

La procédure suivante (`exresignal1`) déclare une exception générale (englobant toutes les exceptions sauf les *warnings* et `NOT FOUND`) qui est définie par un sous-bloc. Dans ce sous-bloc d'instructions, il est prévu de rerouter par `RESIGNAL` tout éventuel signal (en fonction de la variable `v_a`) provenant de l'appel. Dans cet exemple, la procédure déclenche invariablement une erreur car l'instruction `SELECT` concerne une table qui n'existe pas.

À l'appel, si la variable `v_a` est initialisée à 1, la procédure se termine bien (l'erreur est quand même requalifiée en *warning*, visible avec `SHOW WARNINGS`). Il faut que `v_a` soit initialisée à 0 pour que la procédure renvoie l'erreur correcte (ce qui sous-entend que vous devez rerouter le signal). Dans les deux cas `v_passage` vaut 3.

Tableau 7-20 `RESIGNAL` dans un sous-bloc

Procédure	Appels

`CREATE PROCEDURE bdsoutou.exresignal1 ()`
`BEGIN`
`  DECLARE EXIT HANDLER FOR SQLEXCEPTION`
`  /* bloc qui s'exécute après erreur SQL */`
`   BEGIN`
`    SET @v_passage = @v_passage + 2;`
`    IF @v_a = 0 THEN`
`       RESIGNAL;`
`    END IF;`
`  END;`
`  SET @v_passage = @v_passage + 1;`
`  SELECT col FROM table_inexistante;`
`END;`
	`mysql > SET @v_passage = 0$`
	`mysql > SET @v_a = 1$`
	`mysql > CALL exresignal1 ()$`
	`Query OK, 0 rows affected, 1 warning (0.00 sec)`
	
	`mysql > SET @v_passage = 0$`
	`mysql > SET @v_a = 0$`
	`mysql > CALL exresignal1 ()$`
	`ERROR 1146 (42S02): Table 'bdsoutou.table_ inexistante' doesn't exist`

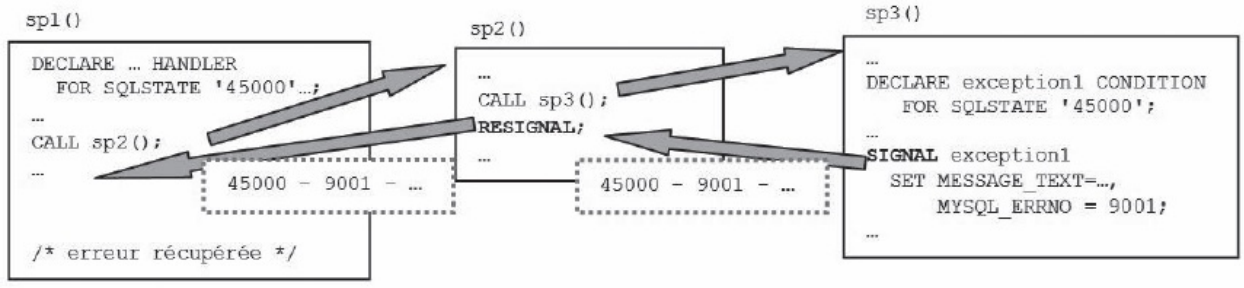
Conclusion : quand on entre dans un sous-bloc, on « perd » le contexte de déclaration des exceptions. Il est donc préférable soit de rerouter le signal dans tous les cas, soit de déclarer des *handlers* locaux qui pourront être différenciés

sans ambiguïté (voir la section « Exceptions »).

RESIGNAL dans un sous-programme

La figure suivante illustre la première forme de l'instruction `RESIGNAL` (à savoir reroutement sans modification du signal). Un premier sous-programme déclare un *handler*, puis appelle un deuxième sous-programme qui reroutera le signal d'un troisième sous-programme qui émet le signal.

Figure 7-6 *Reroutement d'un signal*

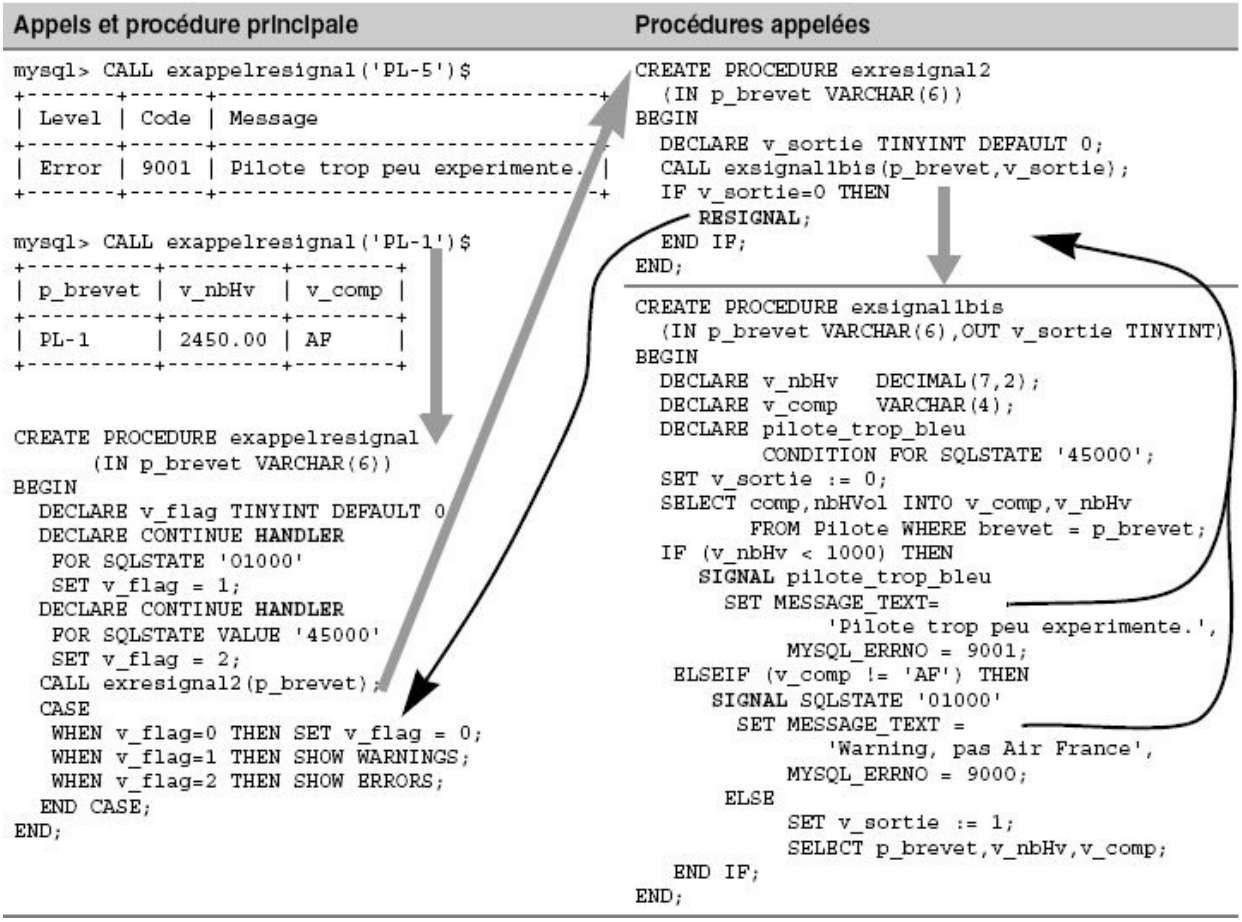


RESIGNAL sans paramètre

La procédure `exappelresignal` appelle la procédure `exsignal2` qui invoque à son tour la procédure `exsignal1`. Trois retours sont possibles : une erreur transmise par `SIGNAL pilote_trop_bleu`…, un *warning* transmis par `SIGNAL SQLSTATE '01000'`… et un code de retour correct (fin du sous-programme sans émission d'un signal). La procédure intermédiaire est chargée de rerouter à la procédure principale le signal initial.

Deux *handlers* sont donc nécessaires dans la procédure principale. La procédure intermédiaire ne doit pas rerouter le signal si le sous-programme le plus imbriqué ne retourne pas d'erreur.

Tableau 7-21 Reroutement inchangé d'un signal



RESIGNAL pour une nouvelle erreur

Utiliser `RESIGNAL` avec des paramètres et la clause `SET` signifie empiler l'exception dans la zone de diagnostics en modifiant l'erreur.

La procédure suivante (`exresignal3`) déclare une exception générale définie par un sous-bloc. Dans ce sous-bloc d'instructions, il est prévu de rerouter par `RESIGNAL` tout signal d'erreur en fonction de la variable `v_a`. La procédure déclenche invariablement une erreur car l'instruction `SELECT` concerne une table qui n'existe pas. Cette erreur est renumérotée 3002.

À l'appel, si `v_a` est initialisée à 1, le reroutement n'est pas effectué et l'erreur est requalifiée en *warning*. Il faut que `v_a` soit initialisée à 0 pour que la procédure reroute l'erreur renumérotée. Dans ce cas, notez l'état de la pile de diagnostics visible avec `SHOW WARNINGS` (la première erreur n'est pas remplacée mais la pile en accueille une nouvelle). Ici le texte n'est pas modifié mais il aurait pu l'être en même temps que le numéro (pour vous en convaincre, ajoutez par exemple `MESSAGE_TEXT='Où est la table?'` à l'instruction `RESIGNAL`).

Tableau 7-22 RESIGNAL pour changer le numéro d'erreur

Procédure	Appels
`DROP TABLE table_inexistante$` `CREATE PROCEDURE exresignal3 ()` `BEGIN` `  DECLARE EXIT HANDLER FOR SQLEXCEPTION` `  BEGIN` `  IF @v_a = 0 THEN` `  **RESIGNAL**` `  SET MYSQL_ERRNO = 3002;` `  END IF;` `  END;` `  SELECT col FROM table_inexistante;` `END;`	`mysql> SET @v_a = 1$` `mysql> CALL exresignal3 ()$` `Query OK, 0 rows affected, 1 warning (0.00 sec)` `mysql> SHOW WARNINGS$` `+-------+------+---------------------------` `--+` `| Level	Code |` `Message |` `+-------+------+---------------------------` `--+` `| Error	1146	Table 'table_inexistante'` `doesn't exist` `mysql> SET @v_a = 0$` `mysql> CALL exresignal3 ()$` `ERROR **3002** (42S02): Table` `'table_inexistante' doesn't` `exist` `mysql> SHOW WARNINGS$` `+-------+------+---------------------------` `---+` `| Level	Code |` `Message |` `+-------+------+---------------------------` `---+` `| Error	1146	Table 'table_inexistante'` `doesn't exist` `| Error	**3002**	Table 'table_inexistante'` `doesn't exist`

RESIGNAL pour une nouvelle condition

Utiliser `RESIGNAL` avec une nouvelle condition signifie empiler l'exception dans la zone de diagnostics en modifiant tout ou partie de l'erreur.

La procédure suivante (`exresignal4`) déclare une exception gérant les codes d'état 42000 définie par un sous-bloc. Dans ce sous-bloc d'instructions, il est prévu de rerouter par `RESIGNAL` toute erreur associée à ce code. La procédure déclenche invariablement une erreur car l'instruction `SELECT` renvoie plusieurs lignes : `ERROR 1172 (42000) : Result consisted of more than one row`. À l'appel, la procédure reroute l'erreur qui est requalifiée en `SQLSTATE` 45000 et renumérotée 3003. Notez l'état de la pile de diagnostics.

Tableau 7-23 RESIGNAL en changeant le code d'état

Procédure	Appels
`CREATE TABLE table_existante(col CHAR(1))$` `INSERT INTO table_existante VALUES ('A')$`	

```
INSERT INTO table_existante VALUES ('A')$

CREATE PROCEDURE exresignal4()
BEGIN
  DECLARE v_col CHAR(1);
  DECLARE EXIT HANDLER FOR SQLSTATE VALUE
'42000'
  BEGIN
    RESIGNAL
      SQLSTATE VALUE '45000'
      SET MYSQL_ERRNO = 3003,
          MESSAGE_TEXT='Il n''y a
personne?';
  END;
  SELECT col INTO v_col FROM
table_existante;
END;
```

```
mysql> CALL exresignal4 ()$
ERROR 3003 (45000): Il n'y a personne ?

mysql> SHOW WARNINGS$
+-------+------+-----------------------+
| Level | Code | Message               |
+-------+------+-----------------------+
| Error | 1172 | Result consisted of
                 more than one row
| Error | 3003 | Il n'y a personne ?
+-------+------+-----------------------+
```

Déclencheurs

Les déclencheurs (*triggers*) n'existent que depuis la version 5 de MySQL. Avant la version 5.0.10, les déclencheurs ne pouvaient même pas accéder aux données de la base. Peter Gulutzan écrivait, en parlant de versions bêta, dans le livre blanc *MySQL 5.0 Triggers* : « Triggers are very new. There are bugs. […] Do not try triggers with a database that has important data in it… »

Bien que bon nombre d'améliorations aient été apportées, les déclencheurs qui modifient des tables n'ont pas encore atteint un niveau de fiabilité suffisant pour la gestion de données sensibles. D'ailleurs, peu d'entre eux doivent être implémentés en production du fait de leur instabilité. Prudence donc avec vos données. Ces limitations, que nous allons détailler, seront sans doute résolues au fur et à mesure des versions majeures du serveur.

Généralités

D'un point de vue général, un déclencheur peut être considéré comme un sous-programme associé à un événement particulier sur la base (action sur une table ou une vue, par exemple). Une table (ou vue) peut « héberger » plusieurs déclencheurs ou aucun. Pour certains SGBD, il existe d'autres types de déclencheurs que ceux associés à une table (ou vue) afin de répondre à des événements qui ne concernent pas les données (exemple : connexion d'un utilisateur particulier, suppression d'une table, démarrage ou arrêt du serveur, déconnexion d'un utilisateur, etc.).

À la différence des sous-programmes, l'exécution d'un déclencheur n'est pas

explicite (par CALL par exemple), c'est l'événement de mise à jour de la table qui lance automatiquement le code programmé dans le déclencheur. On dit que le déclencheur « se déclenche » (l'anglais le traduit mieux : *fired trigger*).

À quoi sert un déclencheur ?

En théorie, un déclencheur permet de :

- Programmer toutes les règles de gestion qui n'ont pas pu être mises en place par des contraintes au niveau des tables. Par exemple, la condition : *une compagnie ne fait voler un pilote que s'il a totalisé plus de 60 heures de vol dans les 2 derniers mois, sur le type d'appareil du vol en question*, ne pourra pas être programmée par une contrainte et nécessitera l'utilisation d'un déclencheur.
- Déporter des contraintes au niveau du serveur et alléger ainsi la programmation client.
- Renforcer des aspects de sécurité et d'audit.
- Programmer l'intégrité référentielle et la réplication dans des architectures distribuées, avec l'utilisation de liens de bases de données.

En théorie, les événements déclencheurs peuvent être :

- une instruction INSERT, UPDATE, ou DELETE sur une table (ou vue). On parle de déclencheurs LMD ;
- une instruction CREATE, ALTER, ou DROP sur un objet (table, vue, index, etc.). On parle de déclencheurs LDD ;
- le démarrage ou l'arrêt de la base (*startup* ou *shutdown*), une erreur spécifique (*not found*, *duplicate key*, etc.), une connexion ou une déconnexion d'un utilisateur. On parle de déclencheurs d'instances.

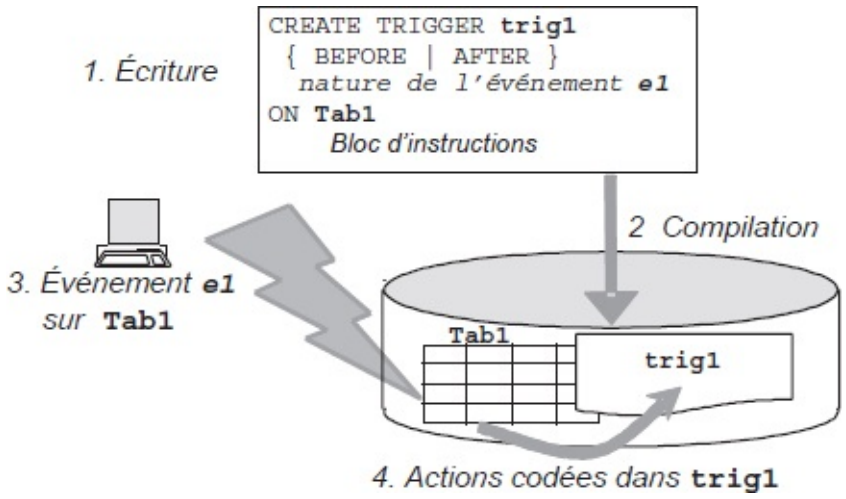
En pratique, MySQL ne permet d'implémenter que les déclencheurs de la première catégorie. Seuls les événements sur les données des tables (ajouts, mises à jour et suppressions) peuvent être interceptés.

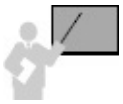
Mécanisme général

Auparavant considérés au niveau *table*, les déclencheurs sont désormais attachés, à plus juste titre, au niveau *database*. En conséquence, le nom d'un déclencheur doit être unique pour chaque base de données. En revanche, différentes bases peuvent héberger un déclencheur de même nom.

La figure suivante illustre les quatre étapes à suivre pour mettre complètement en œuvre un déclencheur (de la création à son test). Il faut d'abord le coder (comme un sous-programme), puis le compiler (il est pour l'instant stocké indépendamment de la table, mais il devrait y être inclus dans les prochaines versions). Par la suite, au cours du temps, chaque événement (qui caractérise le déclencheur) aura pour conséquence son exécution.

***Figure 7-7** Mécanisme des déclencheurs*





Une erreur rencontrée lors de l'exécution d'un déclencheur mis en œuvre sur une table transactionnelle (moteur InnoDB) entraîne l'invalidation de la transaction qui l'a initié (*rollback*). Concernant les tables non transactionnelles (moteur MyISAM, par exemple), l'invalidation n'est pas automatique et les modifications opérées par le déclencheur sont validées…

Par ailleurs, un déclencheur interceptant les ajouts (`INSERT`) sera activé non seulement par chaque instruction `INSERT` rencontrée, mais aussi lors de chaque instruction `LOAD DATA` et `REPLACE` sur la table concernée.

Si vous verrouillez par LOCK TABLES une table associée à un déclencheur, alors les tables qui sont utilisées dans le corps du déclencheur seront également verrouillées.

Avant la version 5.1, pour créer ou supprimer un déclencheur, vous deviez disposer du privilège SUPER. Depuis la version 5.1, le privilège TRIGGER est requis.

Syntaxe

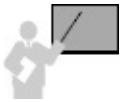
Un déclencheur est composé de deux parties : la description de l'événement traqué et celle de l'action à réaliser lorsque l'événement se produit. La syntaxe de création d'un déclencheur est la suivante :

```
CREATE
    [DEFINER = { utilisateur | CURRENT_USER }]
    TRIGGER nomDéclencheur
   { BEFORE | AFTER } { DELETE | INSERT | UPDATE }
  ON nomTable
  FOR EACH ROW
{ instruction; |
  [etiquette:] BEGIN
       instructions;
   END [etiquette]; }
```

Les options de cette commande sont les suivantes :

- DEFINER précise avec quels droits (ceux de l'utilisateur défini de la forme '*nom'@'serveur'*) s'exécute le déclencheur. Par défaut, les privilèges seront ceux de l'utilisateur qui invoque le déclencheur via un événement (*invoker-rights*). Cette option peut aussi être explicite (DEFINER=CURRENT_USER).
- BEFORE | AFTER précise la chronologie entre l'action à réaliser par le déclencheur LMD et la réalisation de l'événement (BEFORE INSERT exécutera le déclencheur avant de réaliser l'insertion).
- DELETE | INSERT | UPDATE précise la nature de l'événement pour les déclencheurs LMD.
  - Pour DELETE, le déclencheur examine les événements DELETE et REPLACE.
  - Pour INSERT, le déclencheur prend en compte les événements suivants : INSERT, CREATE... SELECT, LOAD DATA, et REPLACE.
  - Pour UPDATE, le déclencheur considère seulement l'événement UPDATE.

- ON nomTable spécifie la table associée au déclencheur LMD.
- FOR EACH ROW différencie les déclencheurs LMD au niveau ligne (le niveau état n'est pas encore pris en charge).
- *instruction* ou *instructions* compose le corps du code du déclencheur.



Vous ne pouvez pas définir plusieurs déclencheurs distincts sur le même événement d'une table (par exemple, « avant d'insérer »). En revanche, il est possible de programmer deux déclencheurs différents concernant la même action générique (par exemple, l'insertion) sur une table (un déclencheur pour BEFORE INSERT et un autre pour AFTER INSERT).

La destruction d'une table (DROP ou TRUNCATE) entraîne la destruction de ses déclencheurs mais ne déclenche aucun *trigger* DELETE.

Étudions à présent plus précisément les caractéristiques du seul type de déclencheur qu'il est actuellement possible de programmer.

Déclencheurs LMD [de lignes]

Pour ce type de déclencheur, l'événement à déterminer est une mise à jour particulière de la base (ajout, modification ou suppression dans une table ou une vue).

L'exécution est, pour l'heure, dépendante du nombre de lignes touchées par l'événement. Seuls les déclencheurs de lignes (*row trigger*) sont permis, car la directive FOR EACH ROW est obligatoire. Ils sont pratiques quand on désire utiliser autant de fois le déclencheur qu'il y a de lignes concernées par une mise à jour.

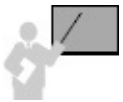
Pour que le déclencheur ne s'exécute qu'une seule fois, quel que soit le nombre de lignes concernées, il faudrait employer un déclencheur d'état (*statement trigger*) qui n'est pas encore reconnu. La directive FOR EACH ROW serait alors absente (ou FOR EACH STATEMENT à sa place), à confirmer dans l'avenir

quand MySQL implémentera cette fonctionnalité.

Dans l'exemple d'une table `t1` ayant cinq enregistrements, si on programme un déclencheur de niveau ligne avec l'événement `AFTER DELETE`, et qu'on lance `DELETE FROM t1`, le déclencheur effectuera cinq fois ses instructions (une fois après chaque suppression). Le tableau suivant explique ce mécanisme :

Tableau 7-24 Exécution d'un déclencheur LMD

Nature de l'événement	État (*statement trigger*) sans `FOR EACH ROW`	Ligne (*row trigger*) avec `FOR EACH ROW`
`BEFORE`	Exécution une fois avant la mise à jour.	Exécution avant chaque ligne mise à jour.
`AFTER`	Exécution une fois après la mise à jour.	Exécution après chaque ligne mise à jour.



Seuls les déclencheurs de lignes peuvent accéder aux anciennes et aux nouvelles valeurs des colonnes de la ligne affectée par la mise à jour prévue par l'événement. Les identificateurs `OLD` et `NEW` sont programmés pour cela. Ce sont des extensions de MySQL à la norme SQL (MySQL suit ainsi Oracle dans ce domaine qui propose `:OLD` et `:NEW`).

Un déclencheur de type `AFTER` ne se lance sur une table *t* que si le déclencheur de type `BEFORE` de la table *t* (s'il existe) et que l'instruction associée à l'événement se sont correctement déroulés.

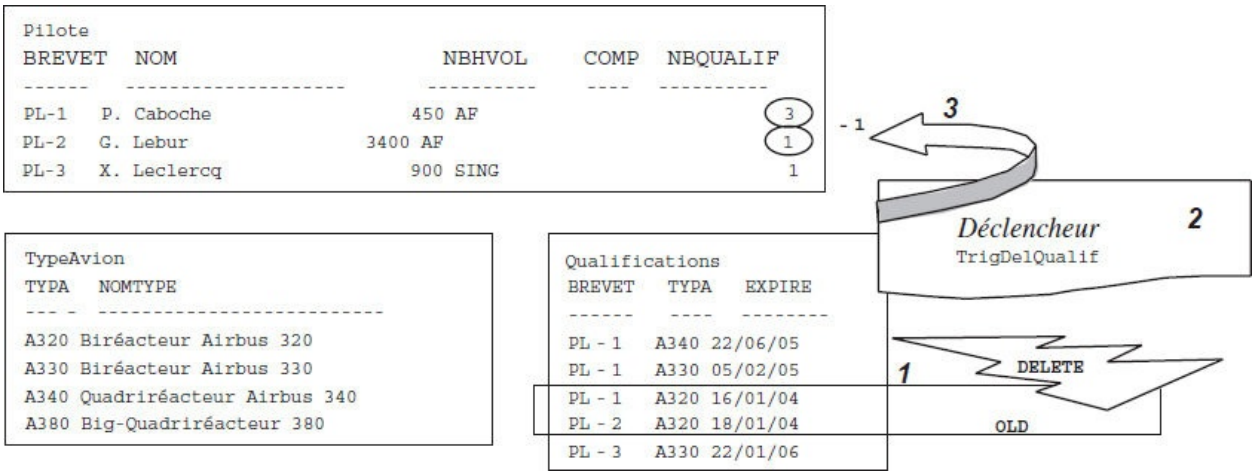
Quand utiliser la directive `OLD` *?*

Chaque enregistrement qui tente d'être supprimé d'une table, qui inclut un déclencheur de type `DELETE FOR EACH ROW`, est désigné par `OLD` au niveau du code du déclencheur. L'accès aux colonnes de ce pseudo-enregistrement dans le corps du déclencheur se fait par la notation pointée.

Considérons l'exemple suivant, et programmons la règle de gestion *tout pilote qui perd une qualification doit voir son compteur automatiquement décrémenter.* Programmons le déclencheur `TrigDelQualif` qui surveille les

suppressions de la table `Qualifications`, et diminue de un la colonne `nbQualif` pour le pilote concerné par la suppression de sa qualification.

***Figure 7-8** Principe du déclencheur* `TrigDelQualif`



L'événement déclencheur est ici `AFTER DELETE`, car il faudra s'assurer que la suppression n'est pas entravée par d'éventuelles contraintes référentielles. On utilise un déclencheur `FOR EACH ROW`, car s'il se produit une suppression de toute la table (`DELETE FROM Qualifications;`), on exécutera autant de fois le déclencheur qu'il y a de lignes détruites.

Chaque enregistrement qui va être supprimé de la table `Qualifications` est désigné par `OLD` au niveau du code du déclencheur. L'accès aux colonnes de ce pseudo-enregistrement dans le corps du déclencheur se fait par la notation pointée.

Le code minimal de ce déclencheur (on ne prend pas en compte le fait qu'il n'existe pas de pilote de ce code brevet) est décrit dans le tableau suivant :

Tableau 7-25 Déclencheur après suppression

Code MySQL	Commentaires

`CREATE TRIGGER TriDelQualif`
` AFTER DELETE ON Qualifications`
` FOR EACH ROW`
	Déclaration de l'événement déclencheur.

`BEGIN`
`  UPDATE Pilote SET nbQualif = nbQualif - 1`
`  WHERE brevet = OLD.brevet;`
`END;`
	Corps du déclencheur.
	Mise à jour du pilote concerné par la suppression.

En considérant les données initiales des tables, le test de ce déclencheur sous l'interface de commande est le suivant. Par ailleurs, la table `Qualifications` ne contient plus que trois enregistrements.

Tableau 7-26 Test du déclencheur `AFTER DELETE`

Événement déclencheur	Résultat
DELETE FROM Qualifications WHERE typa = 'A320';	SELECT * FROM Pilote; +--------+------------+---------+-------+----------+ | brevet | nom | nbHVol | compa | nbQualif | +--------+------------+---------+-------+----------+ | PL-1 | P. Caboche | 450.00 | AF | 2 | | PL-2 | G. Lebur | 3400.00 | AF | 0 | | PL-3 | X. Leclercq | 900.00 | SING | 1 | +--------+------------+---------+-------+----------+

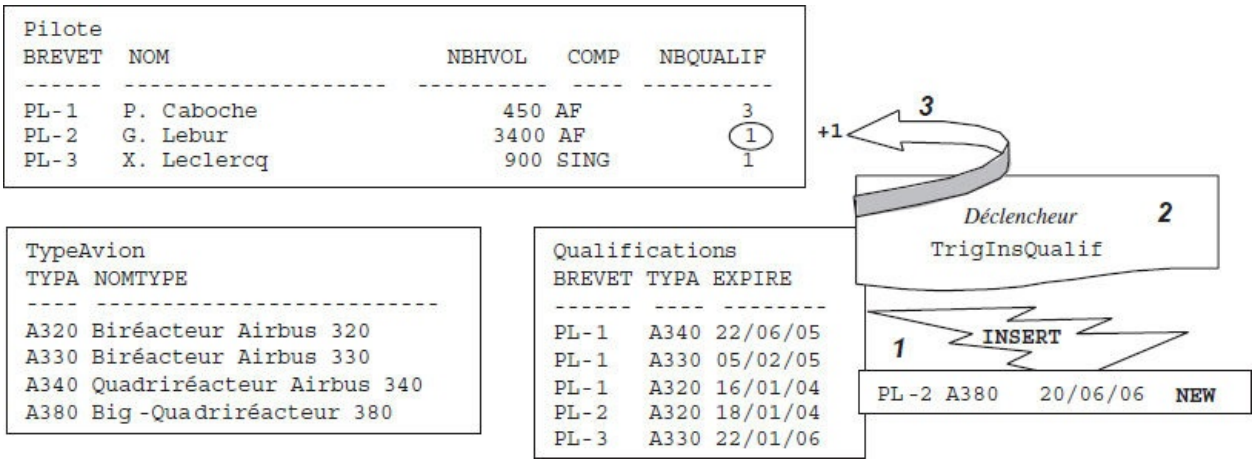
Quand utiliser la directive NEW ?

Chaque enregistrement qui tente d'être ajouté dans la table `Qualifications` est désigné par `NEW` au niveau du code du déclencheur. L'accès aux colonnes de ce pseudo-enregistrement dans le corps du déclencheur se fait par la notation pointée.

Considérons le même exemple et écrivons la règle de gestion *tout pilote qui gagne une qualification doit voir son compteur automatiquement incrémenter*. Programmons le déclencheur `TrigInsQualif` qui surveille les insertions sur la table `Qualifications` et augmente de un la colonne `nbQualif` pour le pilote concerné.

L'événement déclencheur est ici `AFTER INSERT`, car il faudra s'assurer, avant de faire l'insertion, que le code du pilote et celui de l'avion sont corrects (existant dans les tables « père »). On utilise un déclencheur `FOR EACH ROW` car on désire qu'il s'exécute autant de fois qu'il y a de lignes concernées par l'événement déclencheur.

Figure 7-9 *Principe du déclencheur* `TrigInsQualif`



Le code minimal de ce déclencheur (on ne prend en compte aucune erreur potentielle) est décrit dans le tableau suivant :

Tableau 7-27 Déclencheur après insertion

Déclencheur	Commentaires

`CREATE TRIGGER TrigInsQualif`
` AFTER INSERT ON Qualifications`
` FOR EACH ROW`
	Déclaration de l'événement déclencheur.

`BEGIN`
` UPDATE Pilote SET nbQualif = nbQualif + 1`
`  WHERE brevet = NEW.brevet;`
`END;`
	Corps du déclencheur. Mise à jour du pilote concerné par la qualification.

En considérant les données initiales des tables, le test de ce déclencheur (réalisé le 20 décembre 2005) sous l'interface de commande est le suivant :

Tableau 7-28 Test du déclencheur `AFTER INSERT`

Événement déclencheur	Résultat

	`SELECT * FROM Qualifications $`
	`+--------+------+------------+`
	`| brevet | typa | expire |`
	`+--------+------+------------+`
	`| PL-1 | A340 | 2005-06-22 |`
	`| PL-1 | A330 | 2005-02-05 |`
	`| PL-1 | A320 | 2004-01-16 |`
	`| PL-2 | A320 | 2004-01-18 |`
	`| PL-3 | A330 | 2006-01-22 |`

`INSERT INTO`
` Qualifications VALUES`
`('PL-2', 'A380',`
`  SYSDATE());`
	`| PL-2 | A380 | 2005-12-20 |`
	`+--------+------+------------+`
	`SELECT * FROM Pilote$`

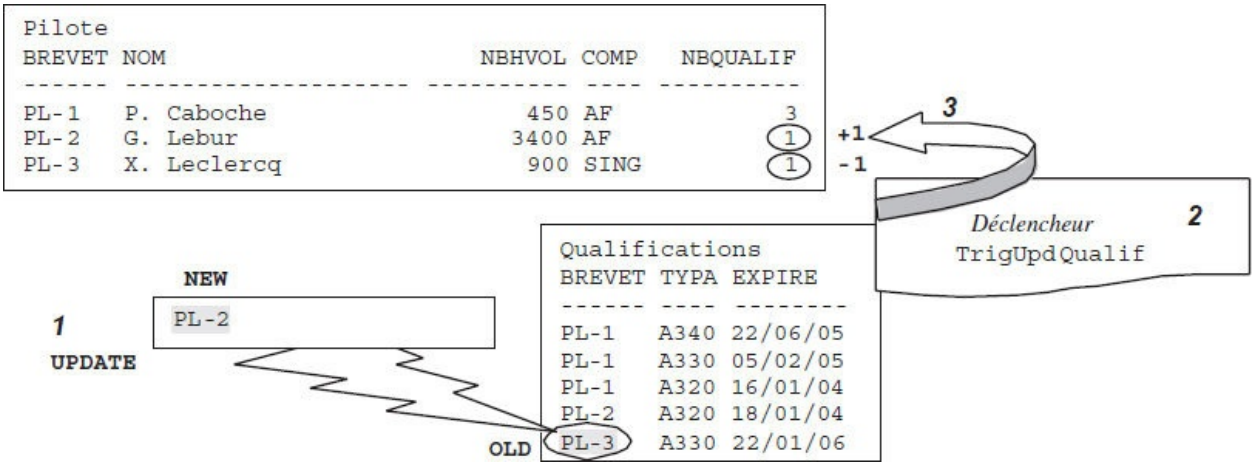
```
+--------+-------------+---------+-------+----------+
| brevet | nom         | nbHVol  | compa | nbQualif |
+--------+-------------+---------+-------+----------+
| PL-1   | P. Caboche  |  450.00 | AF    |        3 |
| PL-2   | G. Lebur    | 3400.00 | AF    |        2 |
| PL-3   | X. Leclercq |  900.00 | SING  |        1 |
+--------+-------------+---------+-------+----------+
```

Quand utiliser à la fois les directives NEW *et* OLD *?*

Seuls les déclencheurs de type UPDATE FOR EACH ROW permettent de manipuler à la fois les directives NEW et OLD. En effet, la mise à jour d'une ligne dans une table fait intervenir une nouvelle donnée qui en remplace une ancienne. L'accès aux anciennes valeurs se fera par la notation pointée du pseudo-enregistrement OLD. L'accès aux nouvelles valeurs se fera par NEW.

La figure suivante illustre ce mécanisme dans le cas de la modification de la colonne brevet du dernier enregistrement de la table Qualifications. Le déclencheur doit programmer deux mises à jour dans la table Pilote.

***Figure 7-10** Principe du déclencheur* TrigUpdQualif



L'événement déclencheur est ici AFTER UPDATE, car il faudra s'assurer que la suppression n'est pas entravée par d'éventuelles contraintes référentielles. Le code minimal de ce déclencheur est décrit dans le tableau suivant :

Tableau 7-29 Déclencheur après modification

Déclencheur	Commentaires
CREATE TRIGGER TrigUpdQualif **AFTER UPDATE ON** Qualifications **FOR EACH ROW** BEGIN	Déclaration de l'événement déclencheur.

`UPDATE Pilote`
`  SET nbQualif = nbQualif + 1`
`  WHERE brevet = NEW.brevet;`
`UPDATE Pilote`
`  SET nbQualif = nbQualif - 1`
`  WHERE brevet = OLD.brevet;`
`END;`
	Corps du déclencheur.
	
	Mise à jour des pilotes concernés par la modification de la qualification.

En considérant les données présentées à la figure précédente, le test de ce déclencheur est le suivant :

Tableau 7-30 Test du déclencheur `AFTER UPDATE`

Événement déclencheur	Résultat
UPDATE Qualifications SET brevet = 'PL-2' WHERE brevet = 'PL-3' AND typa = 'A330'$	SELECT * FROM Pilote$ +--------+-------------+---------+-------+----------+ | brevet | nom | nbHVol | compa | nbQualif | +--------+-------------+---------+-------+----------+ | PL-1 | P. Caboche | 450.00 | AF | 3 | | PL-2 | G. Lebur | 3400.00 | AF | 2 | | PL-3 | X. Leclercq | 900.00 | SING | 0 | +--------+-------------+---------+-------+----------+ SELECT * FROM Qualifications$ +--------+------+------------+ | brevet | typa | expire | +--------+------+------------+ | PL-1 | A340 | 2005-06-22 | | PL-1 | A330 | 2005-02-05 | | PL-1 | A320 | 2004-01-16 | | PL-2 | A320 | 2004-01-18 | | PL-2 | A330 | 2006-01-22 | +--------+------+------------+

Bilan de `NEW` *et* `OLD`

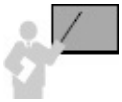
Le tableau suivant résume les valeurs contenues dans les pseudo-enregistrements `OLD` et `NEW` pour les déclencheurs `FOR EACH ROW`. Retenez que seuls les déclencheurs `UPDATE` peuvent manipuler à bon escient les deux types de directives.

Tableau 7-31 Valeurs de `OLD` **et** `NEW`

Nature de l'événement	`OLD.`*colonne*	`NEW.`*colonne*
`INSERT`	Impossible.	Nouvelle valeur.
`UPDATE`	Ancienne valeur.	Nouvelle valeur.
`DELETE`	Ancienne valeur.	Impossible.

MySQL prévient clairement, à la compilation, que vous utilisez une variable OLD dans un déclencheur INSERT – ou NEW dans un déclencheur DELETE – par deux messages de même code, mais de libellés différents, suivant les cas :

- `ERROR 1363 (HY000): There is no NEW row in on DELETE trigger`
- `ERROR 1363 (HY000): There is no OLD row in on INSERT trigger`



Une colonne préfixée de OLD est en lecture seule dans le corps d'un déclencheur.

Une colonne préfixée de NEW ne peut être accessible qu'à l'aide du privilège SELECT associé.

Dans un déclencheur de type BEFORE :

- Il est possible de modifier une colonne préfixée de NEW à la condition de détenir le privilège UPDATE associé. Cela signifie que l'on peut changer un enregistrement avant de l'insérer.
- La valeur d'une colonne AUTO_INCREMENT préfixée par NEW est 0 (et ne suit pas la séquence existante). La valeur actualisée de la séquence ne sera effective que lors de l'insertion.

Attention à ne pas créer de déclencheurs recursifs (exemple d'un déclencheur qui exécute une instruction lançant elle-même le déclencheur, ou deux déclencheurs s'appelant en cascade jusqu'à l'occupation de toute la mémoire réservée).

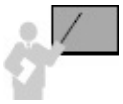
À propos du DEFINER

Les privilèges d'un utilisateur DEFINER différent de celui qui a créé le déclencheur doivent être les suivants :

- le privilège TRIGGER ;
- le privilège SELECT sur la table concernée si le corps du déclencheur contient une directive OLD ou NEW ;

- le privilège `UPDATE` sur la table concernée si une instruction `SET NEW.colonne=…` existe dans le corps du déclencheur ;
- tout autre privilège nécessaire relatif à la manipulation d'autres tables ou vues dans le corps du déclencheur.

Dans le corps d'un déclencheur, la fonction `CURRENT_USER()` est opérationnelle et retourne l'utilisateur `DEFINER` (pas celui qui a initié l'événement déclencheur).



Par ailleurs, le comportement (mode SQL variable système `sql_mode`) dans lequel le déclencheur s'exécutera est celui qui correspond au moment de la création dudit déclencheur. Ainsi, quelle que soit la configuration de cette variable côté serveur, le *trigger* adopte son comportement initial dès que l'événement déclencheur se produit.

Appel de sous-programmes

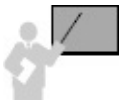
Un déclencheur peut appeler directement par `CALL` (ou dans son corps) un sous-programme MySQL.

Un déclencheur ne peut pas appeler par `CALL` un sous-programme (fonction ou procédure cataloguée) qui retourne des données au client sous forme de traces (avec `SELECT`) ou qui contient du code SQL dynamique (étudié plus loin). Seuls les paramètres d'appel de type `OUT` ou `INOUT` d'un sous-programme permettent de retourner des valeurs après l'appel dans le code du déclencheur.

Par ailleurs, comme tout bon SGBD relationnel qui se respecte, MySQL a bien implémenté le fait qu'un déclencheur ne peut constituer à lui seul une transaction.

En effet, une transaction est constituée d'un ensemble d'instructions devant être validé entièrement ou pas du tout. En d'autres termes, une transaction ne doit pas être validée en partie seulement et un déclencheur s'exécutant au milieu

d'une transaction pourrait bien mener à une telle situation, qui est à éviter. En conséquence, un déclencheur ne doit pas, et ne peut pas, prendre la responsabilité de valider une instruction.



Un déclencheur ne peut pas appeler un sous-programme dont le code contient `START TRANSACTION`, `COMMIT` ou `ROLLBACK` (`ERROR 1422 (HY000): Explicit or implicit commit is not allowed in stored function or trigger`).

Le tableau suivant décrit l'utilisation d'un sous-programme (`procTrigg`) dans un déclencheur (`espionAjoutPilote`) qui s'exécutera avant chaque ajout d'un nouveau pilote. Le sous-programme ajoute simplement une ligne dans la table `Trace`.

```
CREATE PROCEDURE bdsoutou.procTrigg(IN param DATETIME)
BEGIN
  INSERT INTO Trace VALUES
         (CONCAT('Insertion pilote, appel de bdsoutou.procTrigg le ',param));
END;
```

Tableau 7-32 Appel d'un sous-programme dans un déclencheur

Déclencheur	Commentaire

`CREATE TRIGGER bdsoutou.espionAjoutPilote`
` BEFORE INSERT ON Pilote`
` FOR EACH ROW`
`BEGIN`
` CALL bdsoutou.procTrigg(SYSDATE());`
`END;`
	Appel dans le corps du déclencheur d'une procédure MySQL en passant un paramètre d'entrée.

La trace d'exécution en considérant les données initiales des tables est la suivante :

```
mysql> (INSERT INTO Pilote VALUES ('PL-4', 'C. Soutou', 100, 'AF',0)$
Query OK, 1 row affected (0.04 sec)
mysql> SELECT * FROM Pilote$
+--------+------------+---------+-------+----------+
| brevet | nom        | nbHVol  | compa | nbQualif |
+--------+------------+---------+-------+----------+
| PL-1   | P. Caboche |  450.00 | AF    |        3 |
| PL-2   | G. Lebur   | 3400.00 | AF    |        1 |
| PL-3   | X. Leclercq |  900.00 | SING  |        1 |
| PL-4   | C. Soutou  |  100.00 | AF    |        0 |
+--------+------------+---------+-------+----------+
```

```
mysql> SELECT * FROM Trace$
+-------------------------------------------------------------+
| col                                                         |
+-------------------------------------------------------------+
| Insertion pilote, appel de bdsoutou.procTrigg le 2005-12-20 0  |
+-------------------------------------------------------------+
```

Dictionnaire des données

Étudiée au chapitre 5, la base `INFORMATION_SCHEMA` inclut la vue `TRIGGERS` qui renseigne les caractéristiques des déclencheurs qui étaient auparavant considérés au niveau *table*. Ils sont désormais reconnus à juste titre au niveau *database*. Il faut détenir le privilège `SUPER` pour accéder à cette vue.

La requête suivante interroge cette vue et permet de retrouver les noms et les caractéristiques relatives aux événements déclencheurs des trois *triggers* de la base de données `bdsoutou`.

```
SELECT TRIGGER_NAME,EVENT_OBJECT_TABLE "Table",
       EVENT_MANIPULATION "Evenement", ACTION_TIMING,
       EVENT_OBJECT_SCHEMA "Base"
       FROM INFORMATION_SCHEMA.TRIGGERS
       WHERE  TRIGGER_SCHEMA='bdsoutou';
+--------------+----------------+-----------+---------------+----------+
| TRIGGER_NAME  | Table          | Evenement | ACTION_TIMING | Base     |
+--------------+----------------+-----------+---------------+----------+
| TrigInsQualif | Qualifications | INSERT    | AFTER         | bdsoutou |
| TrigUpdQualif | Qualifications | UPDATE    | AFTER         | bdsoutou |
| TriDelQualif  | Qualifications | DELETE    | AFTER         | bdsoutou |
+--------------+----------------+-----------+---------------+----------+
```

Notez que MySQL utilise :

- la colonne `TRIGGER_NAME` pour désigner le nom du déclencheur d'une *database* ;
- la colonne `TRIGGER_SCHEMA` pour désigner le nom de la base de données à laquelle il appartient ;
- les colonnes `EVENT_OBJECT_TABLE` et `EVENT_OBJECT_SCHEMA` pour désigner respectivement le nom de la table qui accueille ce déclencheur ainsi que la base de données qui la contient (elle peut être différente de celle du déclencheur, ici nous raisonnons sur la même) ;
- la colonne `EVENT_MANIPULATION` pour désigner l'événement déclencheur ;
- la colonne `ACTION_TIMING` pour préciser la chronologie de l'événement déclencheur.

La requête suivante interroge cette même vue pour extraire le code du

déclencheur de type *after update* hébergé par la table `Qualifications`, dans la base de données `bdsoutou` :

```
SELECT ACTION_STATEMENT FROM INFORMATION_SCHEMA.TRIGGERS
       WHERE TRIGGER_SCHEMA='bdsoutou' AND EVENT_OBJECT_TABLE='Qualifications'
       AND EVENT_MANIPULATION='UPDATE' AND ACTION_TIMING='AFTER';
+-----------------------------------------------------------------------------+
| ACTION_STATEMENT                                                            |
+-----------------------------------------------------------------------------+
|BEGIN
    UPDATE Pilote SET nbQualif = nbQualif + 1 WHERE brevet = NEW.brevet;
    UPDATE Pilote SET nbQualif = nbQualif - 1 WHERE brevet = OLD.brevet;
 END                                                                          |
+-----------------------------------------------------------------------------+
```

Remarquons que MySQL utilise la colonne `ACTION_STATEMENT` pour contenir le corps du déclencheur (visible aussi par `SHOW TRIGGERS`).

La colonne `ACTION_ORIENTATION` est pour l'instant toujours évaluée à '`ROW`' (déclencheur d'état pas encore opérationnel).

Les colonnes `ACTION_REFERENCE_OLD_ROW` et `ACTION_REFERENCE_NEW_ROW` contiennent, pour l'instant, toujours respectivement '`OLD`' et '`NEW`' (il n'est pas encore possible de renommer ces identificateurs).

Dans le but d'être davantage en phase avec la norme dans les prochaines versions, les colonnes suivantes contiennent, pour l'heure, toujours la valeur `NULL` : `TRIGGER_CATALOG`, `EVENT_OBJECT_CATALOG`, `ACTION_CONDITION`, `ACTION_REFERENCE_OLD_TABLE`, `ACTION_REFERENCE_NEW_TABLE`, et `CREATED`. Les deux premières colonnes sont relatives à la notion de catalogue, la suivante a la possibilité de conditionner un déclencheur (clause `WHEN` d'Oracle), la dernière contiendrait le moment de création du déclencheur.

Programmation d'une contrainte de vérification

Nous avons vu que les contraintes de vérification (`CHECK`) ne sont pas encore prises en charge. Nous avons étudié au chapitre 5 la possibilité d'en programmer à l'aide de vues. Ici, nous allons créer un déclencheur s'en chargeant. Attention, il n'est pas toujours possible d'utiliser un déclencheur pour valider une contrainte de vérification.

Considérons l'exemple du chapitre 5 (*Figure 5-10. Vue simulant la contrainte* CHECK) qui décrit la table Pilote et la contrainte vérifiant qu'un pilote :

- ne peut être commandant de bord qu'à la condition qu'il ait entre 1 000 et 4 000 heures de vol ;
- ne peut être copilote qu'à la condition qu'il ait entre 100 et 1 000 heures de vol ;
- ne peut être instructeur qu'à partir de 3 000 heures de vol.

Le tableau suivant décrit le code du déclencheur. Ici, on choisit de forcer la valeur de la colonne grade pour conserver la cohérence avec les conditions initiales.

Tableau 7-33 Déclencheur simulant un CHECK

Déclencheur	Commentaires
`CREATE TRIGGER TrigInsGrade` `BEFORE INSERT ON Pilote FOR EACH ROW`	Déclaration de l'événement déclencheur.

`BEGIN`
`IF (NEW.grade = 'CDB' AND (NEW.nbHVol<1000)) THEN`
`SET NEW.grade := 'COPI';`
`END IF;`
`IF (NEW.grade = 'CDB' AND (NEW.nbHVol>4000)) THEN`
`SET NEW.grade := 'INST';`
`END IF;`
`IF (NEW.grade = 'COPI' AND (NEW.nbHVol>1000)) THEN`
`SET NEW.grade := 'CDB';`
`END IF;`
`IF (NEW.grade = 'INST' AND (NEW.nbHVol<3000))`
`OR (NEW.nbHVol<100) THEN`
`SET NEW.grade := NULL;`
`END IF;`
`END;`
	Corps du déclencheur.
	
	Test des conditions et mise à jour éventuelle de la nouvelle valeur à insérer au niveau de la colonne grade.

Si aucune condition n'est vérifiée, l'ajout se réalise sans aucun changement. Le test de ce déclencheur est le suivant. On remarque que les quatre premiers INSERT sont inchangés, alors que les deux derniers sont modifiés (mais pas annulés !).

Tableau 7-34 Test du déclencheur BEFORE INSERT

Insertions valides	Insertions non valides
`INSERT INTO Pilote VALUES` `('PL-1','Vielle',1000,'CDB');`	`INSERT INTO Pilote VALUES`

```
INSERT INTO Pilote VALUES
 ('PL-2','Treihlou',450,'COPI');
INSERT INTO Pilote VALUES
 ('PL-3','Filoux',9000,'INST');
INSERT INTO Pilote VALUES
 ('PL-4','Minier',1000,'COPI');
```

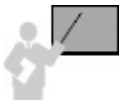
```
INSERT INTO Pilote VALUES
 ('PL-5','Trop jeune',100,'CDB');

INSERT INTO Pilote VALUES
 ('PL-6','Inexperimente',2999,'INST');
```

```
SELECT * FROM Pilote;
+--------+----------------+---------+-------+
| brevet | nom            | nbHVol  | grade |
+--------+----------------+---------+-------+
| PL-1   | Vielle         | 1000.00 | CDB   |
| PL-2   | Treihlou       |  450.00 | COPI  |
| PL-3   | Filoux         | 9000.00 | INST  |
| PL-4   | Minier         | 1000.00 | COPI  |
| PL-5   | Trop jeune     |  100.00 | COPI  |
| PL-6   | Inexperimente  | 2999.00 | NULL  |
+--------+----------------+---------+-------+
```

Programmation dans un déclencheur

Un déclencheur qui se finit bien, c'est-à-dire sans qu'il retourne une erreur ou en déclenche une lui-même, valide l'instruction (l'événement déclencheur) et la transaction appelante se poursuit jusqu'à une éventuelle validation générale (COMMIT) ou invalidation générale ou partielle (ROLLBACK ou ROLLBACK TO…).

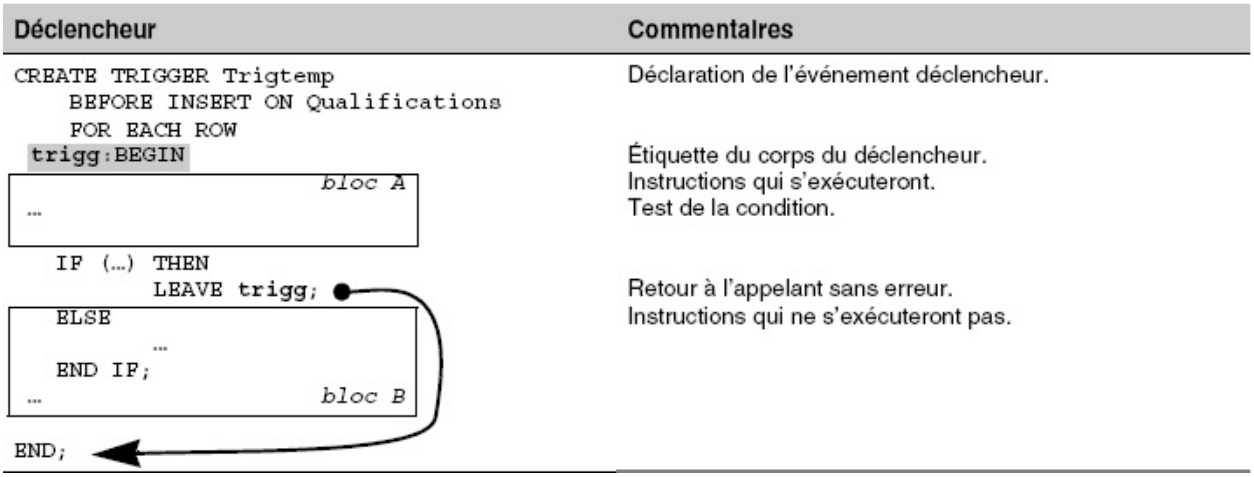


Vous pouvez toutefois interrompre volontairement l'exécution d'un déclencheur. Deux cas sont possibles :

- Si vous désirez valider l'événement déclencheur (sans parcourir tout le code du déclencheur), vous devrez utiliser LEAVE car RETURN n'est pas permis.
- Si vous désirez ne pas valider l'événement déclencheur, vous devrez utiliser une exception (voir paragraphe suivant).

La directive LEAVE permet de sortir d'une boucle ou d'un sous-bloc. Dans les deux cas, vous devrez utiliser une étiquette. L'exemple suivant décrit ce mécanisme sur un déclencheur gérant les ajouts dans la table Qualifications et s'exécutant avant d'ajouter une ligne à la table. L'étiquette est placée ici sur le bloc principal.

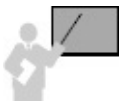
Tableau 7-35 Sortie volontaire d'un déclencheur avec validation



Exceptions dans un déclencheur

Une fonctionnalité importante des déclencheurs consiste à pouvoir invalider l'événement qui a déclenché l'action et retourner un code d'erreur afin que l'appelant ne perde pas la main. Dans tout déclencheur (de type `BEFORE` ou `AFTER`), une erreur lors de l'exécution entraîne l'invalidation de toutes les instructions du bloc et l'arrêt brutal de la transaction.

Jusqu'en version 5.1, une solution consistait à provoquer artificiellement une erreur. Ne choisissez pas une erreur système, par exemple en essayant d'accéder à une table inexistante, mais une erreur applicative comme `NULL` dans une clé primaire. L'inconvénient majeur de cette astuce est que le message d'erreur ne sera jamais explicite, car il ne sera pas en rapport avec la ou les contraintes non satisfaites dans le déclencheur.



Depuis la version 5.5, il est possible de retourner une erreur applicative (numéro d'erreur, code d'état et texte du message) par la directive `SIGNAL`. Ce signal peut être rerouté par `RESIGNAL` jusqu'à l'appelant principal qui peut le traiter à l'aide d'une exception (*handler*).

La solution d'Oracle est le `RAISE_APPLICATION_ERROR`.

Un déclencheur pourra donc dire *non !* à une insertion, à une modification ou à une suppression. Cette programmation ne concerne que les déclencheurs lignes

de type BEFORE, puisqu'il faudra vérifier des conditions dans le corps du déclencheur avant d'accepter l'événement.

L'invalidation d'un déclencheur se traduit par le déclenchement volontaire d'une erreur applicative qui fait avorter l'instruction (INSERT, UPDATE ou DELETE), et par le retour d'une erreur personnalisée (code SQLSTATE, numéro d'erreur et message).

Exemple

Considérons à nouveau l'exemple des qualifications des pilotes et implémentons à l'aide d'un déclencheur la contrainte : *tout pilote ne peut être qualifié sur plus de trois types d'appareils*. L'événement est BEFORE INSERT, car il faudra s'assurer de la condition avant d'autoriser l'ajout d'une nouvelle qualification.

Le déclencheur devra assurer la cohérence entre la valeur de la colonne nbQualif de la table Pilote et les enregistrements de la table Qualifications. Le code de ce déclencheur est décrit dans le tableau suivant. Le signal retourné est de code SQLSTATE 45000 et associé à l'erreur numéro 8000.

Tableau 7-36 Déclencheur avec exception

Déclencheur	Commentaires

`CREATE TRIGGER TrigExceptionInsert`
`BEFORE INSERT ON Qualifications`
`FOR EACH ROW`
	Déclaration de l'événement déclencheur.

`BEGIN`
`DECLARE v_compteur TINYINT(1);`
`SELECT nbQualif`
`INTO v_compteur FROM Pilote`
`WHERE brevet = NEW.brevet;`
	Corps du déclencheur.

`IF v_compteur < 3 THEN`
`UPDATE Pilote SET nbQualif = nbQualif + 1`
`WHERE brevet = NEW.brevet;`
	Test de la condition.

`ELSE`
`SIGNAL SQLSTATE '45000'`
`SET MESSAGE_TEXT =`
`'Le pilote a déjà 3 qualifications',`
`MYSQL_ERRNO = 8000;`
	Retour à l'appelant avec une erreur.

`END IF;` `END;`	

En considérant les données initiales, le test de ce déclencheur est le suivant. La première insertion est valide car le pilote de numéro 'PL-2' n'a initialement qu'une seule qualification. En revanche, la deuxième insertion est invalidée par le déclencheur car le pilote de numéro 'PL-1' a déjà trois qualifications.

```
mysql> INSERT INTO Qualifications VALUES ('PL-2','A380','2010-01-18');
Query OK, 1 row affected (0.01 sec)
mysql> INSERT INTO Qualifications VALUES ('PL-1','A380','2010-01-16');
ERROR 8000 (45000): Le pilote a déjà 3 qualification
```

Récupération d'une erreur d'un déclencheur

Du fait que le signal retourné soit de code SQLSTATE 45000, il suffit de déclarer une exception associée. J'utilise CONTINUE ici mais un autre type de programmation est possible. En effet, si l'événement déclencheur se trouvait dans un bloc, EXIT conviendrait. Pensez aussi à utiliser RESIGNAL dans ce cas.

Tableau 7-37 Récupérer l'erreur d'un déclencheur

Procédure	Commentaires

`CREATE PROCEDURE proc_ajout_trigger`
`(IN v_brevet VARCHAR(6),`
`IN v_typa VARCHAR(4), IN v_expire DATE)`
`BEGIN`	

`DECLARE v_flag TINYINT DEFAULT 0;`
`DECLARE CONTINUE HANDLER`
`FOR SQLSTATE VALUE '45000' SET v_flag = 1;`
	Déclaration de l'exception.

`INSERT INTO Qualifications`
`(brevet,typa,expire)`
`VALUES (v_brevet,v_typa,v_expire);`
	Événement déclencheur.

`CASE`
`WHEN v_flag=0 THEN SET v_flag=0; /* Rien à faire */`
 	
`WHEN v_flag=1 THEN`
`SHOW ERRORS;`
`SELECT nom "C'est lui!"`
`FROM Pilote WHERE brevet=v_brevet;`
`END CASE;`
`END;`
	Traitement de l'erreur.

L'appel de cette procédure qui tente de qualifier un pilote déjà titulaire de trois qualifications est le suivant. L'insertion est invalidée par le déclencheur et l'erreur est récupérée par l'appelant.

```
mysql> CALL proc_ajout_trigger('PL-1','A380','2010-01-16');
+-------+------+----------------------------------+
| Level | Code | Message                          |
+-------+------+----------------------------------+
| Error | 8000 | Le pilote a déjà 3 qualifications |
+-------+------+----------------------------------+
1 row in set (0.00 sec)
+------------+
| C'est lui! |
+------------+
| P. Caboche |
+------------+
```

Tables mutantes

En général, il n'est pas aisé de manipuler la table sur laquelle porte le déclencheur dans le corps du déclencheur lui-même. Alors qu'Oracle parle de *mutating tables* et rend difficile cette programmation, MySQL permet d'accéder à la table en lecture sans problème (par SELECT).

En revanche, il n'est pas possible d'accéder en mise à jour (INSERT, UPDATE ou DELETE) à une table dans le corps d'un déclencheur associé à la table elle-même. L'erreur obtenue est « `ERROR 1442 (HY000): Can't update table 'xxx' in stored function/trigger because it is already used by statement which invoked this stored function/trigger` ».

L'exemple suivant décrit la programmation d'un déclencheur qui compte les lignes d'une table après chaque nouvelle insertion.

Tableau 7-38 Déclencheur (table mutante)

Déclencheur	Trace

`SET @vs_nombre=0$`
`CREATE TRIGGER TrigMutant`
`  AFTER INSERT ON Trace FOR EACH ROW`
`BEGIN`
`  SELECT COUNT(*) INTO @vs_nombre`
`  FROM Trace;`
`END;`
`$`
	`SELECT @vs_nombre$`
	`+------------+`
	`| @vs_nombre |`
	`+------------+`
	`| 0          |`
	`+------------+`
	`INSERT INTO Trace`
	`  VALUES ('Test TrigMutant')$`
	`SELECT @vs_nombre$`
	`+------------+`
	`| @vs_nombre |`
	`+------------+`
	`| 1          |`
	`+------------+`

Restrictions

Pour en finir avec les déclencheurs, je vais terminer la « litanie »…

- Il n'est pas possible de définir un déclencheur sur une table temporaire, une vue ou sur un événement système (connexion, arrêt de la base, etc.).
- Il n'est pas possible de combiner plusieurs événements par 2 ou par 3

(`INSERT OR UPDATE`, `INSERT OR DELETE`, etc.) L'erreur obtenue est : `ERROR 1235 (42000): This version of MySQL doesn't yet support 'multiple triggers`.

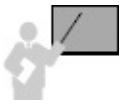
- Il n'est pas possible de désactiver un déclencheur sans le détruire.
- Seul le langage procédural de MySQL peut être utilisé aisément pour coder un déclencheur. Exception faite du C, vous pouvez utiliser la fonction `sys_exec()` disponible sur le site des développeurs communautaires (http://forge.mysql.com/projects/project.php?id=211).
- Toute mise à jour en cascade résultant d'une opération sur une clé primaire (voir le chapitre 2, section « Intégrité référentielle ») ne peut activer un déclencheur.

Suppression d'un déclencheur

Le privilège `TRIGGER` est requis (il s'agissait du privilège `SUPER` avant la version 5.1) pour supprimer un déclencheur. La syntaxe de l'instruction `DROP TRIGGER` est la suivante :

```
DROP TRIGGER [nomBase.]nomDéclencheur;
```

Si le nom de la base est omis, MySQL cherchera à détruire le déclencheur dans la base de données en cours d'utilisation.



Le fait de détruire une table a pour conséquence d'effacer aussi tous les déclencheurs qui lui sont associés.

Le fait de détruire une base supprime toutes les tables. Par conséquent, les déclencheurs passent ainsi tous « à la casserole ».

SQL dynamique

Comme son nom l'indique, la programmation SQL dynamique va permettre d'exécuter des états préparés (*server-side prepared statements*). En d'autres termes, vous aurez la possibilité de construire automatiquement une instruction SQL dans une procédure cataloguée (pas dans une fonction ni un déclencheur).

Une instruction SQL dynamique est stockée en tant que chaîne de caractères et sera évaluée lors de l'exécution et non durant la compilation (à l'inverse d'une instruction statique qui est écrite en « dur » dans un sous-programme).

Il est aussi possible de construire dynamiquement des instructions SQL à partir d'un programme C (*MySQL C API client library*), Java (*MySQL Connector/J*), .Net (*MySQL Connector/NET*) ou PHP par une API écrite en C (*mysqli extension*).

Un état préparé est spécifique à une session (celle où il a été créé). En terminant la session sans désallouer explicitement l'état (nous verrons qu'il y a trois phases pour constuire un ordre, la désallocation étant la dernière), le serveur clôture l'état automatiquement.

Un état préparé est global à la session. En terminant la procédure cataloguée dans laquelle l'état a été créé, l'état n'est pas clôturé. Afin d'éviter d'ouvrir trop d'états simultanément sans les fermer, il est possible d'initialiser la variable système `max_prepared_stmt_count` (0 pour n'avoir qu'un seul état ouvert à tout moment).

Le nombre d'instructions SQL qu'il est possible de construire à considérablement augmenté au fur et à mesure des versions. En version 5.5, vous pouvez notamment (la liste n'est pas exhaustive) utiliser un état préparé pour :

- `ALTER TABLE`, `ANALYZE TABLE` ;
- `CALL`, `COMMIT`, `{CREATE | DROP} DATABASE`, `{CREATE | RENAME | DROP} USER`, `CREATE INDEX`, `CREATE TABLE` ;
- `DELETE`, `DROP INDEX`, `DROP TABLE`, `GRANT`, `INSERT`, `OPTIMIZE TABLE` ;
- `RENAME TABLE`, `REPLACE`, `REVOKE` ;
- `SELECT`, `SET`, `SHOW ...`, `UPDATE`.

Il sera ainsi possible d'extraire des lignes en fonction des colonnes paramétrées, de créer une table ou une base de données de nom variable, etc.

Syntaxe

La construction dynamique d'instructions SQL (*prepared statements*) est basée sur les trois directives suivantes :

```
PREPARE nomEtat FROM étatPréparé;
EXECUTE nomEtat [USING @var1 [, @var2] ...];
{DEALLOCATE | DROP} PREPARE nomEtat;
```

- L'instruction PREPARE associe un nom (insensible à la casse) à une instruction dynamique.
- *étatPréparé* est soit une chaîne soit une variable de session contenant le texte de l'instruction SQL construite (instruction simple, pas d'instructions multiples). Dans cette chaîne, le caractère « ? » (appelé *placeholder*) permet de se substituer à un paramètre.
- L'instruction EXECUTE lance l'ordre paramétré avec éventuellement la clause USING qui reliera les paramètres aux variables de session.
- Pour en terminer avec un ordre préparé, utilisez DEALLOCATE PREPARE qui supprime le contenu de l'ordre (une fin de session désalloue tous les ordres ouverts).

Exemples

Considérons la table Avion contenant deux enregistrements.

```
CREATE TABLE Avion
(immat VARCHAR(6), typeAv CHAR(8), nbHVol DECIMAL(7,2), comp VARCHAR(4));
INSERT INTO Avion VALUES ('F-GLFS', 'A320', 1000, 'AF');
INSERT INTO Avion VALUES ('F-WOWW', 'A380', 1500, 'AF');
```

Instruction DELETE

Le tableau suivant décrit la construction dynamique de l'ordre de suppression des avions dont le nombre d'heures de vol est supérieur à un paramètre spécifié par une variable de session (ici évaluée à 1 000).

Tableau 7-39 Utilisation de DELETE

Code MySQL	Commentaires
`SET @vs_nbhVol  = 1000 $`	Déclaration de la variable de session.

`CREATE PROCEDURE bdsoutou.sousProg()`
`BEGIN`
` PREPARE etat FROM`
`  'DELETE FROM Avion WHERE nbHVol > ?';`
` EXECUTE etat USING @vs_nbhVol;`
` DEALLOCATE PREPARE etat;`
`END;`
	Préparation de l'ordre.
	Exécution.

L'appel (`CALL bdsoutou.sousProg()`) de cette procédure aura pour conséquence de détruire l'avion immatriculé '`F-WOWW`'.

Instruction SELECT

Le tableau suivant décrit la construction dynamique de l'extraction des avions dont le nombre d'heures de vol est égal à un paramètre spécifié par une variable de session (ici évaluée à 1 000). La requête est elle-même stockée dans une variable de session.

Tableau 7-40 Utilisation de `SELECT`

Code MySQL	Commentaires

`SET @vs_chaine =`
`  'SELECT * FROM Avion WHERE nbHVol=?'$`
`SET @vs_nbhVol  = 1000$`
	Déclaration des variables de session.

`CREATE PROCEDURE bdsoutou.sousProg()`
`BEGIN`
` PREPARE etat FROM  @vs_chaine;`
` EXECUTE etat USING @vs_nbhVol;`
` DEALLOCATE PREPARE etat;`
`END;`
	Préparation de l'ordre.
	Exécution.

L'appel de cette procédure aura pour conséquence d'extraire l'avion immatriculé '`F-GLFS`'.

Instruction UPDATE

Le tableau suivant décrit la construction dynamique de l'instruction de modification (augmentation du nombre d'heures de vol d'un pourcentage passé en premier paramètre) d'un avion dont l'immatriculation passe en deuxième paramètre. Notez qu'il n'est pas besoin de doubler le guillemet dans la spécification du deuxième *placeholder* : paramètre `immat` (bien qu'il s'agisse d'une chaîne de caractères).

Tableau 7-41 Utilisation de `UPDATE`

Code MySQL	Commentaires

`SET @vs_chaine =`
`  'UPDATE Avion SET nbHVol=nbHVol*?`
`  WHERE immat=?'$`
`SET @vs_immat  = 'F-GLFS'$`
`SET @vs_pourcent = 1.1$`
	Déclaration des variables de session.

`CREATE PROCEDURE bdsoutou.sousProg()` `BEGIN`	

```
 PREPARE etat FROM  @vs_chaine;
 EXECUTE etat USING
@vs_pourcent,@vs_immat;
 DROP PREPARE etat;
END;
```

Préparation de l'ordre.
Exécution.

L'appel de cette procédure aura pour conséquence d'aumenter de 10 % le nombre d'heures de vol de l'avion immatriculé 'F-GLFS'.

Si l'immatriculation avait été une constante, il aurait fallu doubler le guillemet dans l'affectation de la variable de session :

```
SET @vs_chaine ='UPDATE Avion SET nbHVol=nbHVol*? WHERE immat=''F-GLFS'''$
```

Restrictions

Le *placeholder* (point d'interrogation) d'un état préparé ne peut pas servir à remplacer dans une instruction le nom d'une table, d'une vue, d'un index, d'une colonne ou de tout autre objet MySQL. Ainsi, il sera difficile de construire un ordre dynamique si vous ne connaissez pas a priori les objets que vous allez manipuler.

- Un état préparé ne peut être conçu dans une fonction cataloguée ou un déclencheur.
- Un état préparé ne peut inclure plusieurs instructions SQL même si elles sont séparées par un délimiteur correct.
- Il n'est pas possible d'utiliser un état préparé pour construire un curseur.
- Il n'est pas possible d'inclure un déroutement ou un reroutement (SIGNAL et RESIGNAL) dans un état préparé. Le message d'erreur obtenu est : `ERROR 1295 (HY000): This command is not supported in the prepared statement protocol yet`.

Le *placeholder* ne sert qu'à remplacer des donnés :

- dans la clause WHERE pour un SELECT, UPDATE (dans la clause SET aussi) ou un

DELETE.

- dans la clause VALUES pour un INSERT.

Vous n'aurez pas d'erreur à la compilation de votre procédure mais durant son exécution. MySQL vous retournera soit une erreur au niveau de l'état (ERROR 1210 (HY000): Incorrect arguments to EXECUTE), soit au niveau de l'instruction (ERROR 1064 (42000): You have an error in your SQL syntax; check the manual that corresponds to your MySQL server version for the right syntax to use near...).

Utilisations

Le tableau suivant résume quelques cas permis et les cas non valides associés. N'oubliez pas de doubler chaque guillemet pour affecter une telle chaîne de caractères dans une variable de session.

Tableau 7-42 Utilisation des placeholders

Possibles	Impossibles
SELECT * FROM *table* WHERE *col* = ? ...	SELECT ? FROM *table* WHERE ...
	SELECT * FROM ? WHERE ...
	SELECT * FROM *table* WHERE ? > 1000
INSERT INTO *table* VALUES (?,?,?,?)	INSERT INTO ? VALUES(''F-FRTY'',?,?,?)
UPDATE *table* SET *col*=? WHERE *col* =?...	UPDATE *table* SET ?=? WHERE *col* =? ...
	UPDATE ? SET ...
DELETE FROM *table* WHERE *col* =?...	DELETE FROM *table* WHERE *?* =...
	DELETE FROM ? ...

Afin de pallier un bon nombre de ces limitations, définissez vos instructions dynamiques à l'aide de la fonction CONCAT en incluant éventuellement des *placeholders*. Ainsi, vous construisez une chaîne de caractères dynamiquement compatible avec l'instruction PREPARE de sorte qu'elle soit correctement exécutée.

Exemple sans placeholder

La procédure cataloguée suivante crée dynamiquement, dans la base bdsoutou,

une table de nom passé en premier paramètre. Le nom de la seconde colonne de la table (ici de type INT) est passé en second paramètre de la procédure.

Tableau 7-43 Création dynamique d'une table

Code MySQL	Commentaires

`CREATE PROCEDURE bdsoutou.sousProg`
`  (IN v_param1 VARCHAR(10),`
`   IN v_param2 VARCHAR(10))`
`BEGIN`
`  SET @vs_chaine := CONCAT`
`   ('CREATE TABLE IF NOT EXISTS`
`    bdsoutou.',v_param1,' (immat CHAR(4),`
`'`
`    ,v_param2,' INT)');`
`   PREPARE etat FROM @vs_chaine;`
`   EXECUTE etat;`
`   DEALLOCATE PREPARE etat;`
`END;`
	Construction de la chaîne :
	
	'CREATE TABLE IF NOT EXISTS bdsoutou.v_param1(immat CHAR(4),v_param2 INT)'
	
	Création de la table.

L'appel suivant de cette procédure aura pour effet de créer la table Helico. La commande DESCRIBE confirme la structure de la nouvelle table.

```
CALL bdsoutou.sousProg('Helico','turbine')$
Query OK, 0 rows affected (0.21 sec)
DESCRIBE bdsoutou.Helico $
+---------+---------+------+-----+---------+-------+
| Field   | Type    | Null | Key | Default | Extra |
+---------+---------+------+-----+---------+-------+
| immat   | char(4) | YES  |     | NULL    |       |
| turbine | int(11) | YES  |     | NULL    |       |
+---------+---------+------+-----+---------+-------+
2 rows in set (0.39 sec)
```

Exemple avec placeholder

La procédure cataloguée suivante crée dynamiquement la requête d'extraction du type et du nombre d'heures de vol (colonnes de noms passés en premier et en deuxième paramètres) de la table de nom passé en troisième paramètre, en fonction d'une condition sur une colonne (de nom passé en quatrième paramètre). Cette condition fait intervenir un paramètre (*placeholder*) valant ici 'F-GLFS'.

L'appel de cette procédure avec les paramètres suivants aura pour effet d'extraire les valeurs des deux colonnes du premier enregistrement de la table Avion présentée au début de cette section.

Tableau 7-44 Création dynamique d'une requête avec placeholder

Code MySQL	Commentaires

`CREATE PROCEDURE bdsoutou.sousProg`
`(IN v_param1 CHAR(6), IN v_param2 CHAR(6),`
`IN v_param3 CHAR(5), IN v_param4 CHAR(5))`
`BEGIN`
`SET @vs_immat := 'F-GLFS';`
`SET @vs_chaine :=`
`CONCAT('SELECT ',v_param1,',',v_param2,`
`' FROM bdsoutou.',v_param3,`
`' WHERE ',v_param4,' = ?');`
`PREPARE etat FROM @vs_chaine;`
`EXECUTE etat USING @vs_immat;`
`DEALLOCATE PREPARE etat;`
`END;`
	Avec l'appel suivant, construction de la chaîne :
	`'SELECT typeAv,nbHVol`
	`FROM bdsoutou.Avion`
	`WHERE immat=?'`
	
	Exécution de la requête paramétrée.

```
CALL bdsoutou.sousProg('typeAv','nbHVol','Avion','immat')$
+--------+---------+
| typeAv | nbHVol  |
+--------+---------+
| A320   | 1000.00 |
+--------+---------+
1 row in set (0.01 sec)
Query OK, 0 rows affected (0.01 sec)
```

Paramètres de retour

Depuis la version 5.5.3, un état préparé CALL peut utiliser des paramètres de sortie (OUT et INOUT). Les *placeholders* des précédentes versions étaient limités aux paramètres IN.

L'exemple suivant décrit ce mécanisme avec une procédure qui retourne le type et le prix d'un avion en fonction de son immatriculation. L'appel de la procédure qui a deux paramètres de retour est encapsulé dans un état préparé. Au retour d'appel, les variables de session sont correctement mises à jour.

Tableau 7-45 Sortie volontaire d'un déclencheur avec validation

Procédure	Appel de la procédure
`CREATE PROCEDURE proc_prix_avion` `(IN v_immat VARCHAR(6),` `INOUT v_prix DECIMAL(8,2),` `OUT v_typa CHAR(8))` `BEGIN` `DECLARE v_nbh DECIMAL(7,2);` `SELECT nbHVol,typeAv` `INTO v_nbh, v_typa` `FROM Avion` `WHERE immat=v_immat;` `IF (v_typa='B747') THEN` `SET v_prix=v_prix+v_nbh*10.5;` `ELSE` `SET v_prix=(v_prix*0.9)+v_nbh*15;`	`SET @prixbase = 1000;` `SET @immat = 'F-JUMB';` `SET @type_av = '';` `PREPARE etat FROM 'CALL` `proc_prix_avion(?,?,?)';` `EXECUTE etat USING @immat, @prixbase,` `@type_av;` `SELECT @type_av,@prixbase;` `+----------+-----------+` `| @type_av	@prixbase |` `+----------+-----------+`

```
  END IF;                                  | B747     |  10455.25 |
END;                                       +----------+-----------+
```

Si vous travaillez avec une version antérieure à la 5.5.3, pour simuler des paramètres OUT et INOUT, vous devrez passer en paramètres des variables de session dans le CALL sans rien spécifier dans EXECUTE. Dans notre exemple, il faudrait procéder de la manière suivante :

```
PREPARE etat FROM 'CALL proc_prix_avion(@immat,@prixbase,@type_av)';

EXECUTE etat;
```

Programmation d'événements

Apparue à la version 5.1.6, la programmation d'événements (*events*) permet de planifier dans le temps l'exécution d'un traitement à renouveller éventuellement de façon périodique. On peut assimiler ce mécanisme à celui d'un déclencheur temporel d'état (par opposition au déclencheur ligne qui porte sur une table en particulier). Oracle parle de *job scheduler*, mode de programmation qui permet de répondre à bon nombre de problématiques telles que des sauvegardes quotidiennes, l'exécution de batchs aux heures creuses, l'analyse des tables, etc.

Le contexte

Pour que l'exécution d'un traitement puisse être planifiée, le programmateur d'événements (*event scheduler*) doit être actif. En tant qu'utilisateur, vous devez avoir reçu le privilège SUPER (au niveau global) qui autorise l'exécution de commandes d'administration (CHANGE MASTER TO, KILL, PURGE BINARY LOGS et SET GLOBAL) et de débogage.

Pour activer le programmateur d'événements au niveau d'une session, utilisez SET GLOBAL event_scheduler=1 et, au niveau du serveur, mysqld … --event_scheduler=1. Une fois dans la session, vérifiez que la fonctionnalité est opérationnelle :

```
mysql> SHOW VARIABLES LIKE 'event_scheduler';
+-----------------+-------+
| Variable_name   | Value |
```

```
+-----------------+-------+
| event_scheduler | ON    |
+-----------------+-------+
```

Création d’une planification

Pour définir un événement dans une base, vous devez avoir reçu le privilège EVENT. Le privilège SUPER est dans certains cas requis (cela dépend de la clause DEFINER). Dans un contexte hors réplication, la syntaxe de création d’un événement est la suivante. A minima, les options ON SCHEDULE et DO doivent être renseignées.

```
CREATE [DEFINER = { utilisateur | CURRENT_USER }] EVENT
  [IF NOT EXISTS] [nom_base.]nom_evenement
   ON SCHEDULE AT timestamp [+ INTERVAL intervalle] ...
            | EVERY intervalle
      [STARTS timestamp [+ INTERVAL intervalle] ...]
      [ENDS   timestamp [+ INTERVAL intervalle] ...]
      [ON COMPLETION [NOT] PRESERVE]
   [COMMENT 'commentaire']
   DO instructions_SQL;
```

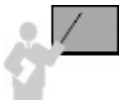
- DEFINER précise avec quels droits (ceux de l’utilisateur défini de la forme `'nom'@'serveur'`) doit s’exécuter le programme événementiel. Par défaut, les privilèges seront ceux de l’utilisateur qui a créé l’événement. Si le créateur n’a pas le privilège SUPER, il n’est pas possible de définir un autre profil d’exécution.
- ON SCHEDULE précise la planification en indiquant l’instant initial (AT ou STARTS), la période de renouvellement (EVERY), puis l’instant final (ENDS). Cette clause peut prendre deux formes :
  - AT timestamp pour une exécution unique (événements non récurrents) ;
  - EVERY intervalle pour répéter l’exécution périodiquement (événements récurrents).
- timestamp désigne un instant au format TIMESTAMP ou DATETIME (précision à la seconde).
- intervalle désigne un délai exprimé sous la forme d’une ou de plusieurs expressions séparées par le signe +, comme suit :

```
nombre {YEAR | QUARTER | MONTH | DAY | HOUR | MINUTE |
        WEEK | SECOND | YEAR_MONTH | DAY_HOUR | DAY_MINUTE |
        DAY_SECOND | HOUR_MINUTE | HOUR_SECOND | MINUTE_SECOND}
```

- ON COMPLETION renseigne sur le fait de supprimer l’événement une fois son

exécution terminée et sa période de renouvellement expirée. Par défaut (NOT PRESERVE), l'événement expiré est détruit. L'option PRESERVE rend persistant l'événement mais inactif si expiré.

- COMMENT pour commenter la planification de l'événement sur 64 caractères.
- DO précède l'instruction SQL ou le bloc d'instructions constituant la programmation de l'événement.



Comme pour les déclencheurs, le mode SQL (sql_mode) dans lequel l'événement s'exécutera est celui qui correspond au moment de la création. De fait, quelle que soit la configuration de cette variable côté serveur, l'événement adopte son comportement initial.

Le tableau suivant présente quelques cas d'utilisation. Si vous utilisez EVERY sans préciser ENDS, vous signifiez que la planification de l'événement est sans fin. S'il manque STARTS, vous signifiez que la planification démarre après la création de l'événement (équivaut à STARTS CURRENT_TIMESTAMP).

Tableau 7-46 Exemples de planifications

Clause ON SCHEDULE…	L'événement s'exécute…
`AT '2011-02-10 10:59:00'`	une seule fois à 10h59 le matin du 10 février 2011.
`AT CURRENT_TIMESTAMP + INTERVAL 1 HOUR`	une seule fois, une heure après sa création.
`AT CURRENT_TIMESTAMP + INTERVAL 3 WEEK + INTERVAL 2 DAY`	une seule fois dans trois semaines et deux jours à la même heure après avoir été créé.
`EVERY 1 DAY STARTS '2011-06-12 04:00:00'`	toutes les nuits à 4h du matin, à partir du 12 juin 2011.
`EVERY 3 MONTH STARTS CURRENT_TIMESTAMP + INTERVAL 1 WEEK`	tous les trois mois, en commençant une semaine après sa création.

`EVERY 12 HOUR STARTS CURRENT_TIMESTAMP + INTERVAL 30 MINUTE ENDS CURRENT_TIMESTAMP + INTERVAL 4 WEEK`	toutes les 12 heures, en commençant 30 minutes après sa création et se terminant au bout de 4 semaines.

Exemple

Considérons la table `Vols` qui recense les réservations de chaque client.

```
+---------+------------+------------+---------------------+
| num_vol | date_vol   | num_client | date_resa           |
+---------+------------+------------+---------------------+
| AF6143  | 2010-10-03 |        170 | 2010-10-03 23:51:52 |
```

Il s'agit de programmer l'événement permettant de suivre minute par minute, et pendant les 2 heures à venir, les réservations effectuées le jour du vol. Le programme va compter toutes les minutes les lignes de cette table vérifiant le numéro de vol et la date (du jour) tout en historisant cette évolution dans la table `Trace` de structure suivante :

```
+---------+---------------------+------------+
| num_vol | heure_comptage      | nombre_pax |
+---------+---------------------+------------+
| AF6143  | 2010-10-03 23:52:06 |          1 |
```

La planification de l'événement démarre dès sa création et cesse 2 heures après. On désire conserver l'événement après son expiration. La trace montre bien que l'exécution a lieu toutes les minutes. Au cours de la première minute, une réservation est comptée, lors de la deuxième minute, deux réservations sont effectuées et, pendant la troisième minute, quatre nouveaux clients se sont inscrits.

Tableau 7-47 Programmation d'un événement minute par minute

Événement et trace de son exécution

```
delimiter $
CREATE EVENT evolution_reservations_AF6143
    ON SCHEDULE
    EVERY 1 MINUTE ENDS CURRENT_TIMESTAMP + INTERVAL 2 HOUR
 ON COMPLETION PRESERVE
 COMMENT 'Comptage des réservations du vol AF6143 pour ce jour'
 DO
   BEGIN
        DECLARE v_numvol CHAR(6) DEFAULT 'AF6143';
        DECLARE v_nbre_pax INT   DEFAULT 0;
```

```
        SELECT COUNT(num_client) INTO v_nbre_pax FROM Vols
           WHERE EXTRACT(DAY FROM date_vol) = EXTRACT(DAY FROM CURRENT_TIMESTAMP)
           AND   EXTRACT(MONTH FROM date_vol) = EXTRACT(MONTH FROM CURRENT_TIMESTAMP)
           AND   EXTRACT(YEAR FROM date_vol) =  EXTRACT(YEAR FROM CURRENT_TIMESTAMP)
           AND   num_vol  = v_numvol;
        INSERT INTO Trace (num_vol,nombre_pax) VALUES (v_numvol,v_nbre_pax);
    END;
$
```

```
mysql> SELECT * FROM Vols;
+---------+------------+------------+---------------------+
| num_vol | date_vol   | num_client | date_resa           |
+---------+------------+------------+---------------------+
| AF6143  | 2010-10-03 |        170 | 2010-10-03 23:51:52 |
| AF6143  | 2010-10-03 |         10 | 2010-10-03 23:52:34 |
| BA450   | 2010-10-03 |        105 | 2010-10-03 23:52:34 |
| AF6143  | 2010-10-03 |        210 | 2010-10-03 23:52:35 |
| AF6143  | 2010-10-03 |         40 | 2010-10-03 23:53:27 |
| AF6143  | 2010-10-03 |        678 | 2010-10-03 23:53:27 |
| AF6143  | 2010-10-03 |        198 | 2010-10-03 23:53:28 |
| AF6144  | 2010-10-03 |        670 | 2010-10-03 23:53:28 |
| AF6143  | 2010-10-03 |        193 | 2010-10-03 23:53:28 |
+---------+------------+------------+---------------------+
9 rows in set (0.00 sec)

mysql> SELECT * FROM Trace;
+---------+---------------------+------------+
| num_vol | heure_comptage      | nombre_pax |
+---------+---------------------+------------+
| AF6143  | 2010-10-03 23:52:06 |          1 |
| AF6143  | 2010-10-03 23:53:06 |          3 |
| AF6143  | 2010-10-03 23:54:06 |          7 |
+---------+---------------------+------------+
```

Bien qu'il soit possible d'utiliser des exceptions et d'appeler une procédure cataloguée (par CALL avec des paramètres sous la forme de constantes) dans le corps de l'événement, il est impossible de passer des paramètres en entrée à un événement.

Dictionnaire des données

La requête suivante interroge la vue EVENTS de la base INFORMATION_SCHEMA et permet de retrouver quelques caractéristiques des événements actifs.

```
SELECT EVENT_SCHEMA, EVENT_NAME, STATUS, LAST_EXECUTED, EVENT_COMMENT
       FROM INFORMATION_SCHEMA.EVENTS
       WHERE STATUS='ENABLED' AND EVENT_TYPE='RECURRING';
+--------------+------------------------------+---------+---------------------+
| EVENT_SCHEMA | EVENT_NAME                   | STATUS  | LAST_EXECUTED       |
```

```
  EVENT_COMMENT                                      |
+-------------+------------------------------+---------+---------------------+
| bdsoutou    | evolution_reservations_AF6143 | ENABLED | 2010-10-04 10:40:42 |
  Comptage des réservations du vol AF6143 pour ce jour   |
+-------------+------------------------------+---------+---------------------+
```

Notez que MySQL utilise :

- Les colonnes EVENT_SCHEMA et EVENT_NAME pour désigner respectivement le nom de la base de données dans laquelle est défini l'événement et l'événement lui-même.
- La colonne STATUS pour indiquer l'état actif ou passif de l'événement (ENABLED ou DISABLED hors réplication).
- Les colonnes LAST_EXECUTED et EVENT_COMMENT pour renseigner respectivement l'instant de la plus récente exécution et le commentaire de l'événement.

Afin de cibler un événement particulier, la requête suivante extrait certaines caractéristiques temporelles et le mode d'exécution d'un événement en particulier.

```
SELECT STARTS, EXECUTE_AT, ENDS, INTERVAL_VALUE, INTERVAL_FIELD,
       SQL_MODE, EVENT_TYPE, EVENT_BODY
       FROM INFORMATION_SCHEMA.EVENTS
       WHERE EVENT_NAME='evolution_reservations_AF6143';
+---------------------+---------------------+-----------------------+
| STARTS              | EXECUTE_AT          | ENDS                  |
| INTERVAL_VALUE      | INTERVAL_FIELD      | SQL_MODE              |
| EVENT_TYPE          | EVENT_BODY          |                       |
+---------------------+---------------------+-----------------------+
| 2010-10-04 09:39:42 | NULL                | 2010-10-04 11:39:42   |
+---------------------+---------------------+-----------------------+
| 1                   | MINUTE              | STRICT_TRANS_TABLES,  |
|                     |                     | NO_AUTO_CREATE_USER,  |
|                     |                     | NO_ENGINE_SUBSTITUTION |
+---------------------+---------------------+-----------------------+
| RECURRING           | SQL                 |                       |
+---------------------+---------------------+-----------------------+
```

Les colonnes suivantes sont mises en œuvre :

- STARTS et ENDS renseignent sur le début et l'arrêt de la planification des événements récurrents.
- EXECUTE_AT renseigne sur le début de la planification des événements non récurrents (ici sans objet).
- INTERVAL_VALUE et INTERVAL_FIELD précisent la fréquence d'exécution.
- SQL_MODE indique le mode d'exécution et EVENT_TYPE précise le caractère récurrent de l'événement (RECURRING) ou non (ONE TIME).

- `EVENT_BODY`, faux ami, ne contient pas le code de l'événement (présent dans `EVENT_DEFINITION`, faux ami aussi) mais le code du langage utilisé dans l'événement (avec la version 5.5, c'est toujours SQL).

Il existe d'autres colonnes à cette vue, citons `DEFINER` (contexte de l'exécution), `ON_COMPLETION` (caractère persistant) et `LAST_ALTERED` (dernière modification).

Modification

La modification permet d'éviter d'avoir à supprimer, puis à recréer un événement pour le planifier à nouveau ou dans d'autres termes. Pour modifier un événement existant dans une base, vous devez avoir reçu le privilège EVENT. Une fois la modification opérée, le profil de l'utilisateur devient le contexte d'exécution (*definer*) de l'événement.

La syntaxe de modification est la suivante. Les options disponibles sont identiques à la création.

```
ALTER [DEFINER={utilisateur|CURRENT_USER}] EVENT
      [nom_base.]nom_evenement
    [ON SCHEDULE planification]
    [ON COMPLETION [NOT] PRESERVE]
    [RENAME TO autre_nom_evenement]
    [ENABLE | DISABLE | DISABLE ON SLAVE]
    [COMMENT 'commentaire']
    [DO instructions_SQL;]
```

Le tableau suivant présente quelques cas d'utilisation de la commande ALTER EVENT.

Tableau 7-48 Modifications d'une planification

Commandes	Conséquences
`ALTER EVENT evolution_reservations_AF6143 DISABLE;`	On désactive temporairement la planification de l'événement (s'il était en cours, il s'arrête).
`ALTER EVENT evolution_reservations_AF6143 ENABLE;`	On réactive la planification de l'événement (s'il était arrêté, il repart éventuellement en fonction des options STARTS et ENDS existantes).

`delimiter $`
`ALTER EVENT evolution_reservations_AF6143`
`DO`
`BEGIN`
`…`
`SELECT COUNT(num_client)`
`INTO v_nbre_pax FROM Vols`
	Modification du programme de l'événement. Ici, on compte les réservations de la

```
      WHERE EXTRACT(DAY FROM date_vol) =
        EXTRACT(DAY FROM CURRENT_TIMESTAMP)-1
    AND  …
 END;
$
```

veille. Le nouveau code s'exécutera dès la prochaine planification.

Restrictions actuelles

Quelques restrictions sont encore d'actualité, certaines peuvent se résoudre par une programmation système ou à l'aide de SQL dynamique.

Un événement ne peut pas être défini pour se déclencher à la fin d'un autre événement.

Si deux événements sont créés avec la même planification, il n'y a pas de moyen de donner une quelconque priorité ou un ordre d'exécution à l'un des deux événements.

Inutile d'essayer de tracer avec `SHOW` ou `SELECT` des variables dans un événement. Dans des termes Unix, la sortie va tout droit vers `/dev/null`.

Gestion de XML

La gestion de XML a commencé en version 5.1. Les deux seules fonctions (`ExtractValue` et `UpdateXML`) qui étaient proposées alors sont toujours limitées. Les versions majeures du serveur qui ont succédé n'ont pas apporté de fonctionnalités supplémentaires. La documentation renvoie toujours à un forum interne, dont l'activité laisse toujours à désirer. Bref, XML ne semble pas être une priorité de MySQL.

Ce chapitre présente d'une part les fonctions permettant la gestion de documents XML à travers des fonctionnalités XPath et d'autre part l'instruction `LOAD XML` qui permet de charger du contenu XML en base.

Fonctions XML

À ce jour, les seules fonctions qui permettent de manipuler des fragments de documents XML en utilisant des prédicats sont `ExtractValue`, qui lit un document XML, et `UpdateXML`, qui permet de modifier un document XML.

Extraction de contenu simple (ExtractValue)

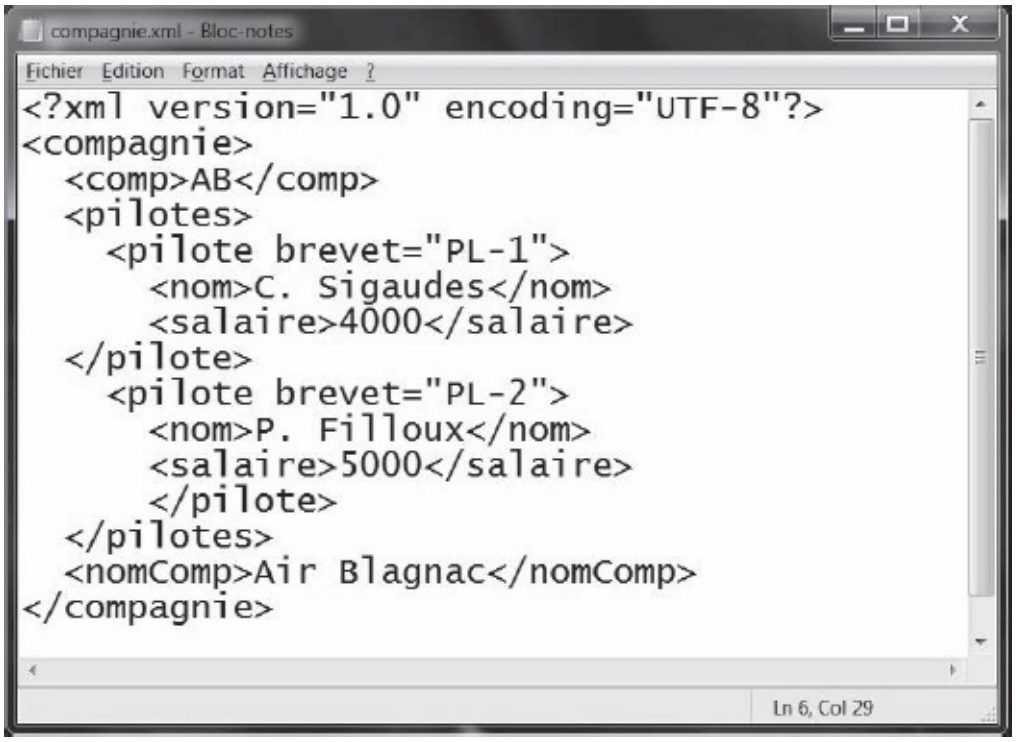
La fonction `ExtractValue` dispose de deux paramètres : le premier désigne le fragment XML à considérer, le second une expression XPath (aussi appelée *locator*). Une valeur textuelle (`CDATA`) est retournée, laquelle correspond au premier nœud fils vérifiant l'expression XPath (équivaut à la fonction `text()` appliquée au nœud en question).

Tableau 7-49 Fonction XML

Fonction	Utilisation
`ExtractValue(string xml_frag, string xpath_expr)`	Extraction d'une valeur à partir d'un fragment XML respectant un prédicat XPath.

Considérons le document XML suivant (`compagnie.xml`) et cherchons à extraire différents contenus (au sein d'éléments ou d'attributs respectant des prédicats simples) tels que le code de la compagnie, le nom du premier pilote, le numéro du second, etc.

Figure 7-11 *Document initial XML*



Il est possible d'utiliser une variable de session (ou de procédure) qui contiendra le document XML que vous désirez analyser. Dans le cas de la variable de session, l'affectation est la suivante et ne doit pas contenir le prologue du document.

```
SET @var_xml =
  '<compagnie>
     <comp>AB</comp>
     …
   </compagnie>';
```

Si vos documents XML contiennent des espaces de noms (*XML namespaces*), MySQL ne les prendra pas en compte. Il faut donc les enlever avant d'initialiser la variable de travail…

Le tableau suivant présente quelques extractions d'éléments et d'attributs XML par le biais d'expressions et de fonctions XPath.

Tableau 7-50 Requêtes sur un document XML

Besoins	Requêtes et résultats

Code de la compagnie	`mysql> SELECT ExtractValue(@var_xml,'/compagnie/comp');`
	`+----------+`
	`| AB |`
	`+----------+`

Nom du premier pilote	`mysql> SELECT ExtractValue(@var_xml, '/compagnie/pilotes/pilote[1]/nom');`
	`+----------+`
	`| C. Sigaudes |`
	`+----------+`

Numéro du deuxième pilote (attribut)	`mysql> SET @j =2;`
	`mysql> SELECT ExtractValue(@var_xml, '/compagnie/pilotes/pilote[$@j]/@brevet');`
	`+-------+`
	`| PL-2 |`
	`+-------+`

Nom du premier et du deuxième pilote	`mysql> SELECT ExtractValue(@var_xml, '/compagnie/pilotes/pilote[1]/nom | /compagnie/pilotes/pilote[2]/nom');`
	`+----------+`
	`| C. Sigaudes P. Filloux |`
	`+----------+`

Salaire du pilote ayant pour nom 'P. Filloux'	`mysql> SELECT ExtractValue(@var_xml, '/compagnie/pilotes/pilote[nom=''P. Filloux'']/salaire');`
	`+------+`
	`| 5000 |`
	`+------+`

Nom des pilotes ayant un salaire inférieur à 4 500 €	`mysql> SELECT ExtractValue(@var_xml, '/compagnie/pilotes/pilote[salaire<4500]/nom');`
	`+----------+`
	`| C. Sigaudes |`
	`+----------+`

Masse salariale des pilotes (fonction XPath)	`mysql> SELECT ExtractValue(@var_xml, 'sum(/compagnie/pilotes/pilote/salaire)');`
	`+------+`
	`| 9000 |`
	`+------+`

Moyenne des salaires des pilotes (fonctions XPath)	`mysql> SELECT ExtractValue(@var_xml, 'sum(/compagnie/pilotes/pilote/salaire) div count(/compagnie/pilotes/pilote)');`
	`+------+`
	`| 4500 |`
	`+------+`

Nom des pilotes ayant un salaire supérieur à 4 500 € et dont le prénom commence par la lettre 'P'	`mysql> SELECT ExtractValue(@var_xml, '/compagnie/pilotes/pilote[salaire>4800 and contains(nom,''P.'')]/nom');`
	`+----------+`
	`| P. Filloux |`
	`+----------+`

Il est possible d'utiliser une variable de procédure pour extraire des informations XML. La procédure suivante extrait le nom de tous les pilotes en considérant le document XML à travers la variable `var_xml`.

Tableau 7-51 Procédure cataloguée manipulant un document XML

Code de la procédure	Résultat

`DELIMITER $`
 	
`CREATE PROCEDURE myproc()`
`BEGIN`
`DECLARE i INT DEFAULT 1;`
`DECLARE var_xml VARCHAR(500) DEFAULT`
`'<compagnie>`
`  <comp>AB</comp>`
`  <pilotes><pilote brevet="PL-1">`
`      <nom>C. Sigaudes</nom>`
`      <salaire>4000</salaire>`
`  </pilote>`
`    <pilote brevet="PL-2">`
`      <nom>P. Filloux</nom>`
`      <salaire>5000</salaire>`
`    </pilote></pilotes>`
`  <nomComp>Air Blagnac</nomComp>`
`</compagnie>';`
`WHILE i < 2 DO`
` SELECT ExtractValue(var_xml,`
`      '/compagnie/pilotes/pilote/nom[$i]');`
` SET i = i+1;`
`END WHILE;`
`END $`
	`mysql> DELIMITER ;`
	`mysql> CALL myproc;`
	`+--------------------------------------+`
	`| ExtractValue(var_xml,`
	`  '/compagnie/pilotes/pilote/nom[$i]')  |`
	`+--------------------------------------+`
	`| C. Sigaudes P. Filloux               |`
	`+--------------------------------------+`

La seule façon de réaliser une extraction multiple de plusieurs éléments (ou attributs) consiste à utiliser une variable de procédure cataloguée qui contiendra le document, et à parcourir l'arborescence XML via cette variable dans une boucle.

Remplacement de contenus (UpdateXML)

La fonction `UpdateXML` dispose de trois paramètres. Le premier désigne le document XML à traiter, le deuxième contient l'expression XPath (*locator*) qui localise le fragment à remplacer, et le troisième désigne le fragment à insérer. À la suite de cette fonction, le document XML final est retourné (une fois l'ancien fragment remplacé par le nouveau).

Tableau 7-52 Fonctions XML

Fonction	Utilisation
`UpdateXML(string xml_target, string`	Remplace un fragment XML par un

xpath_expr, string *new_xml*)	nouveau fragment qui est retourné par la suite.

Considérons à nouveau le document XML initial (la compagnie et ses deux pilotes) affecté à la variable `@var_xml` afin de présenter quelques exemples de mises à jour.

Tableau 7-53 Modifications de fragments XML

Besoins	Requêtes et résultats

Modification de la racine du document	`mysql> SELECT UpdateXML(@var_xml, '/compagnie', '<resultat>C\'est la crise... </resultat>');`
	`+----------------------------------------+`
	`| <resultat>C'est la crise...</resultat> |`
	`+----------------------------------------+`

Remplacement de l'élément `nom` (terminal) du premier pilote par un nouvel élément	`mysql> SELECT UpdateXML(@var_xml, '/compagnie/pilotes/pilote[1]/nom', '<remplace_par>D. Daurat</remplace_par>');`
	`+----------------------------------------+`
	`| <compagnie>`
	`  <comp>AB</comp>`
	`  <pilotes>`
	`  <pilote brevet="PL-1">`
	`  <remplace_par>D. Daurat</remplace_par>`
	`  <salaire>4000</salaire>`
	`  </pilote>`
	`  <pilote brevet="PL-2">`
	`  <nom>P. Filloux</nom>`
	`  <salaire>5000</salaire>`
	`  </pilote>`
	`  </pilotes>`
	`  <nomComp>Air Blagnac</nomComp>`
	`</compagnie>  |`
	`+----------------------------------------+`

Remplacement du premier élément `pilote` (composé) par un nouvel élément	`mysql> SELECT UpdateXML(@var_xml, '/compagnie/pilotes/pilote[1]', '<remplace_par>D. Daurat</remplace_par>');`
	`+----------------------------------------+|`
	`| <compagnie>`
	`  <comp>AB</comp>`
	`  <pilotes>`
	`  <remplace_par>D. Daurat</remplace_par>`
	`  <pilote brevet="PL-2">`
	`  <nom>P. Filloux</nom>`
	`  <salaire>5000</salaire>`
	`  </pilote>`
	`  </pilotes>`
	`  <nomComp>Air Blagnac</nomComp>`
	`</compagnie>  |`
	`+----------------------------------------+`

	`mysql> SELECT UpdateXML(@var_xml, '/compagnie/nomComp', '<nomComp>JJ-Mercier Airlines</nomComp>');`
	`+----------------------------------------+`
	`| <compagnie>`

Remplacement de la valeur d'un élément (`nomcomp`)	<comp>AB</comp> <pilotes> <pilote brevet="PL-1"> <nom>C. Sigaudes</nom> <salaire>4000</salaire> </pilote> <pilote brevet="PL-2"> <nom>P. Filloux</nom> <salaire>5000</salaire> </pilote> </pilotes> <nomComp>**JJ-Mercier Airlines**</nomComp> </compagnie>

Du fait qu'il n'existe aucune fonction dédiée à l'insertion et à la suppression d'éléments XML (ou d'attributs), vous devrez programmer vos mises à jour au sein d'une procédure cataloguée (une requête ou un bloc ne suffiront pas). Si vous souhaitez :

- Ajouter un nouvel élément : remplacez le précédent par lui-même suivi du nouveau.
- Modifier un élément (ou attribut) : remplacez l'élément en entier.
- Supprimer un élément : remplacez-le par l'élément nul (chaîne vide).

Le tableau suivant présente quelques mises à jour en adoptant ces principes. Les modifications concernent le contenu XML présent dans la variable `@var_xml`.

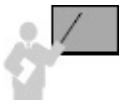
Tableau 7-54 Modifications de fragments XML par réécriture

Besoin	Procédure cataloguée
Ajout d'un troisième pilote	CREATE PROCEDURE proc_XML_test () BEGIN SELECT UpdateXML(@var_xml,'/compagnie/pilotes/pilote[2]', '<pilote brevet="PL-2"> <nom>P. Filloux</nom> <salaire>5000</salaire> </pilote> **<pilote brevet="PL-3"> <nom>A. Rami</nom> <salaire>5400</salaire> </pilote>**') INTO @var_xml; END $
Modification d'un attribut (le	CREATE PROCEDURE proc_XML_test () BEGIN SELECT UpdateXML(@var_xml,'/compagnie/pilotes/pilote[2]',

numéro de brevet) du deuxième pilote	'<pilote brevet="**PL-2new**"> <nom>P. Filloux</nom> <salaire>5000</salaire> </pilote>') INTO @var_xml; SELECT @var_xml; END $
Suppression du deuxième pilote	CREATE PROCEDURE proc_XML_test () BEGIN SELECT UpdateXML(@var_xml, '/compagnie/pilotes/pilote[2]','') INTO @var_xml; END $

Afin de répercuter la modification (effectuée sur la variable de session) durablement dans un fichier, utilisez l'option `INTO OUTFILE` en précisant le chemin et le nom du fichier qui contiendra le nouveau document XML. Ainsi, l'instruction suivante écrit, dans le fichier `sortie.xml` situé dans le répertoire `C:\Donnees\dev\xml-mysql`, le contenu du document XML modifié préalablement.

```
SELECT @var_xml INTO OUTFILE 'C:\\Donnees\\dev\\xml-mysql\\sortie.xml';
```



Si le chemin XPath ne retourne aucun contenu (ou, au contraire, s'il en retourne plusieurs), la fonction `UpdateXML` renvoie le fragment XML présent en entrée (inchangé donc).

Ainsi, à titre d'exemples, les expressions suivantes ne modifieront pas le contenu XML en entrée (contenu ici dans la variable `@var_xml`).

Tableau 7-55 Résultats d'un chemin XPath

Ne retourne aucun contenu	Retourne plusieurs contenus
/* chemin incorrect */ SELECT UpdateXML(@var_xml, '**/compagnie/pilote**', '<resultat>…</resultat>')	/* il existe plusieurs pilotes */ SELECT UpdateXML(@var_xml, '**/pilotes/pilote**', '<resultat>…</resultat>');
/* il existe moins de 4 pilotes */ SELECT UpdateXML(@var_xml, '**/compagnie/pilotes/pilote[4]**', '<resultat>…</resultat>');	/* il existe plus d'un pilote */ SELECT UpdateXML(@var_xml, '**//pilote**', '<resultat>…</resultat>');

Gestion des erreurs

Selon l'expression incorrecte XML ou XPath rencontrée, les fonctions

ExtractValue et UpdateXML retourneront soit une erreur, soit un *warning*.

Expression XPath incorrecte

L'erreur 1105 est retournée par les fonctions ExtractValue et UpdateXML lorsque l'expression XPath est invalide. Le tableau suivant présente une extraction et une modification dont l'expression XPath est incorrecte.

Tableau 7-56 Expression XPath incorrecte

Instructions incorrectes	Résultat identique

`SELECT ExtractValue('<pilote">`
`  <nom>C. Sigaudes</nom>`
`  <salaire>4000</salaire>`
`  </pilote>','/&brevet');`
 	
`SELECT UpdateXML(`
`'<pilote">`
`  <nom>C. Sigaudes</nom>`
`  <salaire>4000</salaire>`
`  </pilote>',  '/&brevet',`
`  '<resultat>D. Daurat</resultat>');`
	`ERROR 1105 (HY000): XPATH syntax error: '&brevet'`

Fragments mal formés

Lorsque le fragment XML passé en paramètre (pas le second paramètre de UpdateXML mais le premier) n'est pas bien formé, la valeur NULL est retournée et un *warning* est généré. La commande SHOW WARNINGS permet de restituer le message d'erreur. Le tableau suivant présente une extraction et une modification contenant le même fragment mal formé et restituant le même *warning*.

Tableau 7-57 Fragments XML mal formés

Fragment mal formé	Résultat
`SELECT ExtractValue('<pilote>` `  <nom>C. Sigaudes</nom>` `  </pilote></fin>','//nom');` `SELECT UpdateXML('<pilote>` `  <nom>C. Sigaudes</nom>` `  </pilote></fin>', '/pilote/nom',` `  '<resultat>D. Daurat</resultat>');`	`mysql> SHOW WARNINGS;` `+---------+------+----------------------` `----+` `| Level	Code |` `Message |` `+---------+------+----------------------` `----+` `| Warning	1525	Incorrect XML value:` `'parse` ` error at line 3 pos 18: '</fin>'` ` unexpected (END-OF-INPUT` `wanted)' |` `+---------+------+----------------------` `----+`

MySQL ne contrôle pas la forme du fragment à remplacer avec `UpdateXML`. Ainsi, un document mal formé peut être retourné.

Limitations

MySQL est encore bien loin de ses principaux concurrents en ce qui concerne la gestion de XML. De nombreuses limites brident encore les fonctions `ExtractValue` et `UpdateXML`.

Extraction de nœuds

La fonction `ExtractValue` ne retourne que du contenu XML textuel (`CDATA`) et n'est pas capable d'extraire un nœud XML avec ses composants (seul le contenu terminal d'un élément ou d'un attribut peut être retourné).

Afin d'illustrer cette limitation, considérons les extractions présentées au tableau suivant.

- La première ne retourne que le contenu situé directement sous l'élément sélectionné par l'expression XPath, les sous-éléments sont ignorés.
- La seconde extraction ignore le dernier pilote car son contenu est complexe. Seuls les contenus qui sont terminaux sous les balises `pilote` sont extraits.

Tableau 7-58 Extraction d'éléments composés

Contenu mixte	Avec l'utilisation de //

`mysql> SELECT ExtractValue(`
`  '<compagnie>`
`  <pilotes>les noms suivent`
`  <pilote>C. Sigaudes</pilote>`
`  <pilote>P. Filloux</pilote>`
	`mysql> SELECT ExtractValue(`
	`  '<compagnie>`
	`  <pilotes>les noms suivent`
	`  <pilote>C. Sigaudes</pilote>`
	`  <pilote>P. Filloux</pilote>`
	`  <pilote><nom>B. Meleton</nom>`

```
    </pilotes>
   </compagnie>'
 ,'/compagnie/pilotes') AS
contenu_pilotes;
+------------------------------+
| contenu_pilotes              |
+------------------------------+
| les noms suivent             |
+------------------------------+
```

```
</pilote>
   </pilotes>
  </compagnie>'
,'//pilote') AS contenu_pilote;
+----------------------+
| contenu_pilote       |
+----------------------+
| C. Sigaudes P. Filloux |
+----------------------+
```

Les élements vides

Aucune distinction n'est faite par `ExtractValue` entre le cas où aucun fragment ne convient à l'expression et celui où c'est un élément vide qui convient. La valeur `NULL` est retournée dans les deux cas.

Il en va de même si le fragment n'est pas bien formé.

Afin de pallier cette limitation, vous devez inclure la fonction XPath `count` dans l'expression *XPath*. Le tableau suivant présente, dans la colonne de gauche, deux requêtes qui n'utilisent pas `count` et renvoient `NULL`, alors qu'il ne s'agit pas du même document XML en entrée. Dans la colonne de droite, la requête qui utilise `count` permet de connaître l'existence d'éléments vides.

Tableau 7-59 Utilisation de la fonction Xpath `count`

Requêtes incorrectes	Requête correcte

`/* il n'y a pas de balise pilote */`
`SELECT ExtractValue(`
`  '<compagnie>`
`  <pilotes></pilotes>`
`  </compagnie>'`
`  ,'/compagnie/pilotes/pilote');`
`/* 2 éléments pilotes vides sont présents */`
`SELECT ExtractValue(`
`  '<compagnie>`
`  <pilotes>`
`  <pilote/>`
`  <pilote/>`
`  </pilotes>`
`  </compagnie>'`
`  ,'/compagnie/pilotes/pilote');`
	`/* 2 éléments pilote vides sont présents */`
	
	`mysql> SELECT ExtractValue(`
	`  '<compagnie>`
	`  <pilotes>`
	`  <pilote/>`
	`  <pilote/>`
	`  </pilotes>`
	`  </compagnie>'`
	`  ,'count(/compagnie/pilotes/pilote)');`
	
	`+--------------------------------+`
	`| 2                              |`
	`+--------------------------------+`

Le tableau suivant présente un exemple qui remplace la racine (donc le document entier) par un fragment XML plus que mal formé.

Tableau 7-60 Remplacement par un document mal formé

Remplacement	Résultat
`mysql> SELECT UpdateXML(@var_xml, '/compagnie', '<racine> <nimporte>hein?</quoi>') AS xml_pb;`	`+--------------------------------------+ | xml_pb | +--------------------------------------+ | <racine> <nimporte>hein?</quoi> | +--------------------------------------+`

Fin de la litanie…

Il n'est pas possible d'utiliser une variable au sein d'une expression XPath dans un paramètre de la fonction `ExtractValue` (bogue #32911).

Bon nombre de fonctions XPath ne sont pas supportées : `id`, `lang`, `local-name`, `name`, `number`, `namespace-uri`, `normalize-space`, `starts-with`, `string`, `substring-after` `substring-before` et `translate`.

On ne peut pas comparer des nœuds entre eux au sein d'une expression XPath (via la notation abrégée ou à l'aide des directives `preceding-sibling` ou `following-sibling`).

Les entités (incluant les caractères spéciaux) XML ne sont pas reconnues.

Chargement XML [LOAD XML]

Mécanisme analogue à `LOAD DATA INFILE` étudié au chapitre 2, l'importation de données au format XML peut être programmée à l'aide de la commande `LOAD XML`. Le privilège `FILE` est requis.

Syntaxe

La commande `LOAD XML` lit un fichier situé dans un répertoire du serveur (ou du client) et insère des lignes dans une table (à créer au préalable). La syntaxe de

cette commande est la suivante :

```
LOAD XML [LOW_PRIORITY | CONCURRENT] [LOCAL]
   INFILE 'cheminEtNomFichier'
     [REPLACE | IGNORE]
     INTO TABLE [nomBase.]nomTable
     [CHARACTER SET nomJeuCaracteres]
     [ROWS IDENTIFIED BY '<balise>']
     [IGNORE nb [LINES | ROWS]]
     [(colonne_ou_variable ,...)]
     [SET colonne = expression ,...];
```

- `LOW_PRIORITY` diffère l'exécution de la commande tant qu'il existe un client qui accède à la table. Cela ne concerne que les moteurs de stockage qui verrouillent au niveau table (`MyISAM`, `MEMORY` et `MERGE`).
- `CONCURRENT` permet de jouer la transparence avec des tables `MyISAM` mais cette option dégrade les performances.
- `LOCAL` précise que le document XML est lu sur le poste client, puis envoyé au serveur. Le chemin peut être absolu ou relatif. En l'absence de cette option, le document XML est situé sur le serveur (chemin absolu, relatif ou dans le répertoire de la base).
- `INFILE` désigne le nom du document XML ainsi que son chemin dans l'arborescence des fichiers.
- `REPLACE` remplace systématiquement les anciens enregistrements par les nouveaux (valeur de clé primaire ou d'index unique).
- `CHARACTER SET` associe un jeu de caractères (`latin1`, `ascii`…) temporairement (durée de la commande).
- `ROWS IDENTIFIED BY` précise le nom de l'élément XML qui définira chaque ligne de la table.
- `IGNORE nb` permet d'ignorer les *nb* premiers éléments du document XML.

Exemple

Supposons que nous disposions du document XML (`avions.xml`) situé dans le répertoire `C:\Donnees\dev\xml-mysql` et de la table `tab_avions_xml`. La colonne `creation` est ajoutée à titre d'exemple.

Tableau 7-61 Document XML existant et table à créer

Document XML (avions.xml)	Création de la table
`<?xml version="1.0" encoding="ISO-8859-1"?>`	

```
<avis>
 <avi immat="F-GWSD" cap="100" typeav="A320">
</avi>
 <avi immat="F-GWSS" cap="50"  typeav="Concorde"/>
 <avi immat="F-FGLS" cap="4"   typeav="TB20">
</avi>
 <avi immat="F-GAFO" cap="490" typeav="A380"/>
 <avi immat="F-GFDG" cap="110" typeav="A320"/>
</avis>
```

```
CREATE TABLE tab_avions_xml
 (immat  CHAR(6) NOT NULL PRIMARY
KEY,
  cap      SMALLINT NOT NULL,
  typeav   VARCHAR(20) NOT NULL,
  creation TIMESTAMP);
```

Le tableau suivant présente, d'une part, la commande d'importation du document XML et, d'autre part, l'extraction de la table `tab_avions_xml` fraîchement mise à jour.

Tableau 7-62 Document XML existant et table à créer

Commande d'importation	Extraction de la table

`LOAD XML LOCAL INFILE 'C:\\Donnees\\dev\\xml-mysql\\avions.xml' REPLACE INTO TABLE tab_avions_xml ROWS IDENTIFIED BY '<avi>';`	`mysql> SELECT * FROM tab_avions_xml;`
	`+--------+-----+----------+---------------------+`
	`| immat  | cap | typeav   | creation            |`
	`+--------+-----+----------+---------------------+`
	`| F-FGLS |   4 | TB20     | 2010-09-02 23:25:23 |`
	`| F-GAFO | 490 | A380     | 2010-09-02 23:25:23 |`
	`| F-GFDG | 110 | A320     | 2010-09-02 23:25:23 |`
	`| F-GWSD | 100 | A320     | 2010-09-02 23:25:23 |`
	`| F-GWSS |  50 | Concorde | 2010-09-02 23:25:23 |`
	`+--------+-----+----------+---------------------+`

Formats d'entrée

La structure du document XML que vous désirez importer doit respecter un des trois formats suivants (valable également pour la commande `mysqldump`). L'exemple précédent respectait le premier format (toutes les données se trouvaient dans les attributs).

Tableau 7-63 Formatages du document XML

Document XML	Élément XML
Les attributs composent les colonnes et les données sont les valeurs des attributs.	*<balise col1="valeur1" col2="valeur2" …/>*

Les sous-éléments composent les colonnes, et les données sont les valeurs de ces sous-éléments.	<balise> <col1>valeur1</col1> <col2>valeur2</col2> … </balise>
Les valeurs des attributs (name) des sous-éléments field composent les colonnes, et les données sont les valeurs de ces sous-éléments.	<balise> <field name='col1'>valeur1</field> <field name='col2'>valeur2</field> … </balise>

Concernant notre exemple, les deux autres formats d'importation (seul un élément est décrit) auraient également été valides.

Tableau 7-64 Formats valides

Type d'écriture	Élément « ligne » du document XML
Éléments pour chaque colonne	<avi> <immat>F-GWSD</immat> <cap>100</cap> <typeav>A320</typeav> </avi>
Sous-éléments field pour chaque colonne	<avi> <field name="immat">F-GWSD</field> <field name="cap">100</field> <field name="typeav">A320</field> </avi>

Colonnes et variables

Les options suivantes de l'instruction LOAD XML permettent de ne pas respecter l'ordre des champs du document XML et celui des colonnes de la table. La liste de la première option peut contenir des noms de colonnes ou des variables. La clause SET permet d'affecter des valeurs à des colonnes dans une certaine mesure.

Le tableau suivant présente quelques importations qu'il est possible de réaliser à partir du document XML initial (contenu dans le fichier avions_xml).

- La première importation précise que seules les valeurs des champs (éléments ou attributs suivant le mode de stockage) typeav, immat et cap seront extraits du document XML.
- La deuxième importation met à jour la colonne jour et augmente de 20 % les capacités pour chaque avion.
- La troisième importation met à jour la colonne cap via une sous-requête

présente dans la clause SET qui doit retourner une valeur scalaire et ne pas interroger la table en chargement.

- La quatrième importation ajoute des colonnes à la table par l'intermédiaire de variables non valuées.
- La dernière importation décrit le comportement du chargement lorsque des colonnes de la table ne correspondent pas aux données XML.

Tableau 7-65 Importation de documents XML

Création de la table et importation	Contenu de la table après importation

`CREATE TABLE avions_xml_2`
`(typeav VARCHAR(20) NOT NULL,`
`immat CHAR(6) NOT NULL PRIMARY KEY,`
`cap SMALLINT NOT NULL);`
 	
`LOAD XML LOCAL INFILE`
`'C:\\Donnees\\dev\\xml-mysql\\avions.xml'`
`REPLACE INTO TABLE tab_avions_xml_2`
`ROWS IDENTIFIED BY '<avi>'`
`(typeav, immat, cap);`
	`mysql> SELECT * FROM tab_avions_xml_2;`
	`+----------+--------+-----+`
	`| typeav | immat | cap |`
	`+----------+--------+-----+`
	`| TB20 | F-FGLS | 4 |`
	`| A380 | F-GAFO | 490 |`
	`| A320 | F-GFDG | 110 |`
	`| A320 | F-GWSD | 100 |`
	`| Concorde | F-GWSS | 50 |`
	`+----------+--------+-----+`

`CREATE TABLE tab_avions_xml_3`
`(immat CHAR(6) NOT NULL PRIMARY KEY,`
`cap DECIMAL,`
`typeav VARCHAR(20) NOT NULL,`
`jour DATE);`
 	
`LOAD XML LOCAL INFILE`
`'C:\\Donnees\\dev\\xml-mysql\\avions.xml'`
`REPLACE INTO TABLE tab_avions_xml_3`
`ROWS IDENTIFIED BY '<avi>'`
`(immat, cap, typeav)`
`SET jour=SYSDATE(), cap=cap*1.2;`
	`mysql> SELECT * FROM tab_avions_xml_3;`
	`+--------+------+----------+------------+`
	`| immat | cap | typeav | jour |`
	`+--------+------+----------+------------+`
	`| F-FGLS | 5 | TB20 | 2010-09-03 |`
	`| F-GAFO | 588 | A380 | 2010-09-03 |`
	`| F-GFDG | 132 | A320 | 2010-09-03 |`
	`| F-GWSD | 120 | A320 | 2010-09-03 |`
	`| F-GWSS | 60 | Concorde | 2010-09-03 |`
	`+--------+------+----------+------------+`

`LOAD XML LOCAL INFILE`
`'C:\\Donnees\\dev\\xml-mysql\\avions.xml'`
`REPLACE INTO TABLE tab_avions_xml_2`
`ROWS IDENTIFIED BY '<avi>'`
`SET cap = (SELECT MAX(cap)`
`FROM tab_avions_xml`
`WHERE typeav='A320');`
	`mysql> SELECT * FROM tab_avions_xml_2;`
	`+----------+--------+-----+`
	`| typeav | immat | cap |`
	`+----------+--------+-----+`
	`| TB20 | F-FGLS | 110 |`
	`| A380 | F-GAFO | 110 |`
	`| A320 | F-GFDG | 110 |`
	`| A320 | F-GWSD | 110 |`
	`| Concorde | F-GWSS | 110 |`
	`+----------+--------+-----+`

`LOAD XML LOCAL INFILE`
`'C:\\Donnees\\dev\\xml-mysql\\avions.xml'`
`REPLACE INTO TABLE tab_avions_xml_3`
`ROWS IDENTIFIED BY '<avi>'`
`(immat, @var_cap, typeav, @var_date);`
	`mysql> SELECT * FROM tab_avions_xml_3;`
	`+--------+------+----------+------+`
	`| immat | cap | typeav | jour |`
	`+--------+------+----------+------+`
	`| F-FGLS | NULL | TB20 | NULL |`
	`| F-GAFO | NULL | A380 | NULL |`
	`| F-GFDG | NULL | A320 | NULL |`
	`| F-GWSD | NULL | A320 | NULL |`
	`| F-GWSS | NULL | Concorde | NULL |`
	`+--------+------+----------+------+`

	`mysql> SELECT * FROM tab_avions_xml_4;`

```
CREATE TABLE tab_avions_xml_4
   (immat    CHAR(6) NOT NULL PRIMARY KEY,
    typeav   VARCHAR(20) NOT NULL,
    couleur  VARCHAR(30),
    nbHVol   DECIMAL,
    jour     DATE);

LOAD XML LOCAL INFILE
'C:\\Donnees\\dev\\xml-mysql\\avions.xml'
   REPLACE INTO TABLE tab_avions_xml_4
   ROWS IDENTIFIED BY '<avi>';
```

```
+--------+----------+---------+--------+---
---+
| immat  | typeav   | couleur | nbHVol |
jour |
+--------+----------+---------+--------+---
---+
| F-FGLS | TB20     | NULL    |   NULL |
NULL |
| F-GAFO | A380     | NULL    |   NULL |
NULL |
| F-GFDG | A320     | NULL    |   NULL |
NULL |
| F-GWSD | A320     | NULL    |   NULL |
NULL |
| F-GWSS | Concorde | NULL    |   NULL |
NULL |
+--------+----------+---------+--------+---
---+
```

La commande `LOAD XML` n'est pas autorisée dans les procédures cataloguées (`ERROR 1314 LOAD XML is not allowed in stored procedures`).

Les variables ne peuvent pas être initialisées avant ou calculées durant le chargement du fait du typage inexistant à la déclaration.

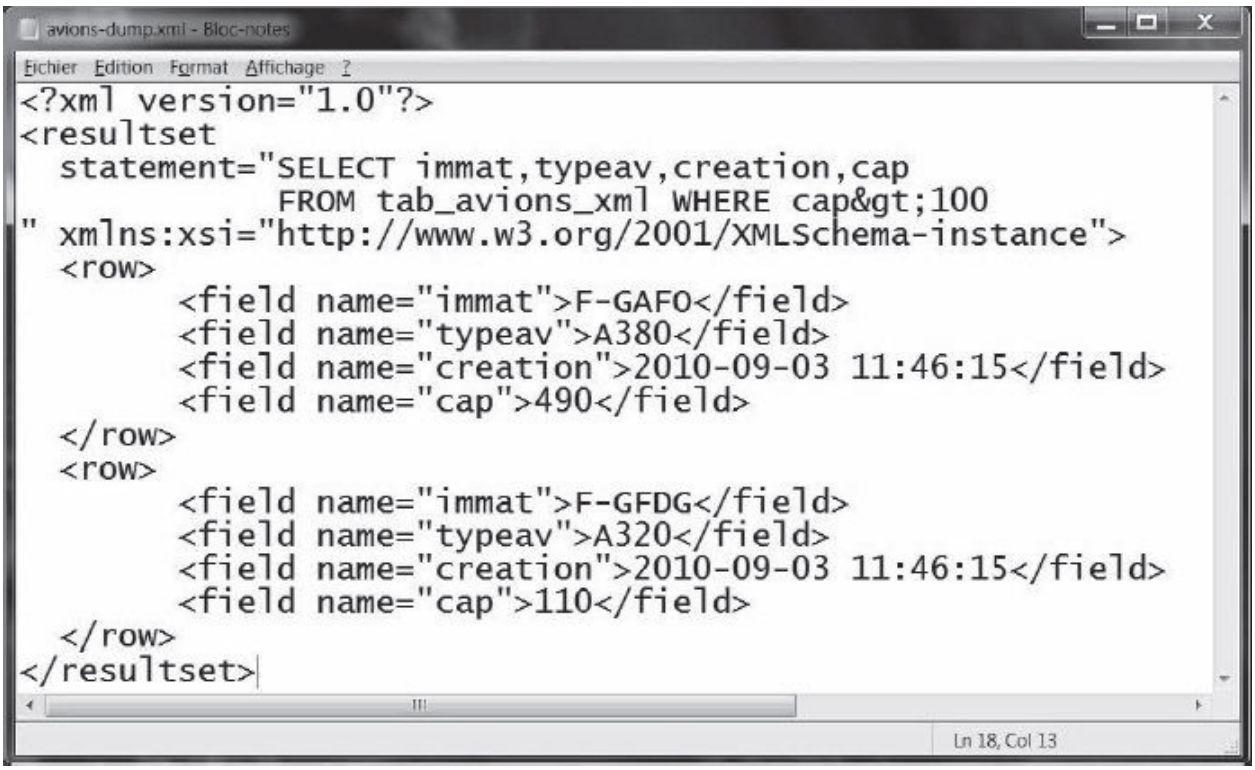
Un dump en ligne de commande

L'opération inverse de `LOAD XML` consiste à exporter, sous la forme d'un document XML, le résultat d'une extraction (en utilisant l'option `--xml` abordée à la fin du chapitre 4). La commande suivante s'exécute au niveau d'une ligne de commande (ici Windows). Les options `user`, `password` et `database` sont à renseigner. L'option `--e` est nécessaire : elle permet l'exécution de la commande, puis quitte MySQL.

```
mysql --user=… --host=localhost --password=… --database=…
      --xml -e
      "SELECT immat,typeav,creation,cap
              FROM tab_avions_xml WHERE cap>100"
      > C:\Donnees\dev\xml-mysql\avions-dump.xml
```

La table est désormais exportée et le document XML généré respecte le troisième des formats valides (chaque colonne est définie par un élément `field`).

Figure 7-12 *Document XML exporté*



Les données JSON sont prises en compte depuis la version 5.7.8. Vous pouvez les exploiter dans de simples colonnes (voir la section « Fonctions SQL pour JSON »), ou dans des collections si vous envisagez d'utiliser l'envirronnement Document Store.

Gestion de JSON

MySQL depuis 5.7

Basées sur la nouvelle interface de commandes *MySQL Shell* depuis la version 5.7.12, de nouvelles méthodes permettent d'accéder à des collections de documents JSON. Ces documents sont stockés dans des tables et sont accessibles à l'aide de JavaScript, Python ou SQL. Ce mode de programmation qui est appelé *Document Store* par MySQL préfigure les avancées de traitements « NoSQL ». Par ailleurs, bon nombre de fonctions (toutes préfixées par `JSON_` comme `JSON_INSERT` par exemple) sont maintenant dédiées au type `JSON` pour manipuler des documents dans des tables traditionnelles.

Mise en place de l'environnement Document Store

Le protocole *X Protocol* qui s'installe à l'aide de X Plugin, permet à un client (qui implémente X DevAPI) d'exécuter des opérations de type CRUD en mode document. Cet environnement permet aussi d'exécuter des opérations SQL, après une authentification de type SASL.

Vous devez disposer de l'option *X Plugin* pour bénéficier du *X Protocol* et de *MySQL Shell*. Depuis la version 5.7.12, une fenêtre vous le propose à l'installation (celle qui va suivre) et dans tous les cas il n'y a aucun téléchargement particulier à opérer. Si votre installation est déjà réalisée, vous devez migrer dans un premier temps dans une version 5.7.12 (ou postérieure).

Si vous utilisez Windows, lancez l'installeur, choisissez « Reconfigure » au niveau du serveur puis cochez la case d'activation du protocole X au niveau de la quatrième fenêtre de l'assistant.

Figure 7-13 *Installation du protocole X*

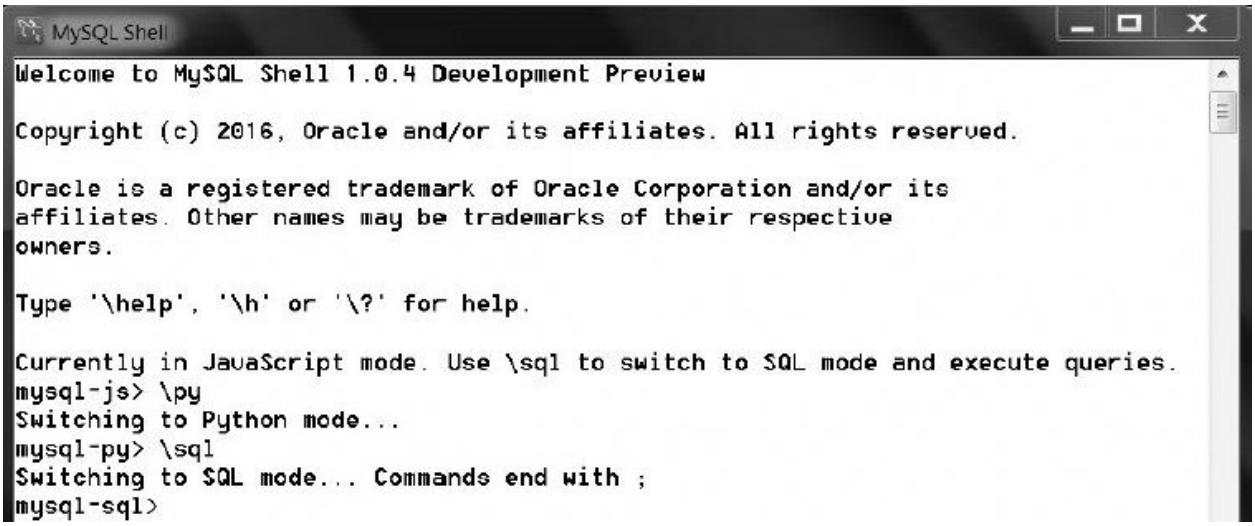
Par la suite, vous devrez installer *MySQL Shell* en téléchargeant un installateur (paquetage msi) sur http://dev.mysql.com/downloads/shell/.

Pour les autres utilisateurs (Linux et Mac), il faudra installer en premier lieu *MySQL Shell* puis *X Plugin* dans une fenêtre de commandes `mysql (INSTALL PLUGIN mysqlx SONAME 'mysqlx.so';)`. Une fois l'opération terminée, vous devrez vérifier la présence de cette nouvelle interface de commandes (`mysqlsh`).

La nouvelle interface vous permet de programmer dans les trois modes

(JavaScript par défaut, Python ou SQL) ; le prompt indique dans quel mode sera interprétée toute commande. Pour les utilisateurs de Windows, pensez à ajouter le chemin (par défaut `C:\Program Files\MySQL\MySQL Shell x.y\bin`) à la variable d'environnement *Path*. Enfin, l'installation de *X Plugin* crée l'utilisateur local nommé *mysqlxsys* qui est verrouillé.

Figure 7-14 *Interface de commandes MySQL Shell*



Les documents JSON

Un document JSON (*JavaScript Object Notation*) permet de structurer des informations comme en XML, mais de manière moins verbeuse. En revanche, un document JSON, bien formé ou pas, ne dispose pas de contrôle d'une grammaire. Les champs sont de type atomique, tableaux (entre crochets) ou listes (entre accolades). Dans l'exemple suivant, le document est constitué de nombreuses informations positionnées à différents niveaux de l'arborescence.

Tableau 7-66 Document JSON

Document JSON	Structure des champs

```
{
    "_id" : "FE345_90",
    "num_vol" :"AF6143",
    "date_vol" :"2016-07-19",
   "desc_vol" :
   {"porte" : "20A",
    "plan_vol" : {
     "depart" : "21 :45",
     "parking" : "3R2",
```

```
  "SID" : "AGN-2W",
  "STAR" : "TOU-5T",
  "arrivee" : null},
"sieges" :
 [{"prenom" : "Guy",
   "nom" :    "Blanchet",
   "siege" :  "15A"},
  {"prenom" : "Gerard",
   "nom" :    "Diffis",
   "siege" :  "15B"},
  {"prenom" : "Victor",
   "nom" :    "Ferrage",
   "siege" :  "15C"}]
  }
 }
```

Les collections JSON

Une collection est un container qui permet de stoker et de manipuler des documents JSON. Depuis la version 5.7.8, le type natif JSON permet de ne plus stocker les données en tant que caractères mais à l'aide d'un format binaire plus efficace.

Création

Ces documents sont stockés dans la colonne `doc` d'une table dont la structure est fortement contrainte (voir le tableau suivant) et dont le nom constitue celui de la collection. Une fois la collection créée, il sera possible de bénéficier des opérations CRUD : `add`, `find`, `modify` et `remove`. Il est inutile de valider (par `commit`) ces opérations, le shell s'en charge. D'ailleurs, l'instruction de validation retourne une erreur si on l'utilise dans cet environnement.

Tableau 7-67 Déclaration d'une collection JSON

Création de la table	Commentaires
	La clé primaire de la table doit se nommer `_id` pour

```
CREATE TABLE tab_vol_json
  (_id VARCHAR(32)
      GENERATED ALWAYS AS
      (JSON_UNQUOTE(JSON_EXTRACT(doc, '$._id')))
      STORED PRIMARY KEY,
   doc JSON)
ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

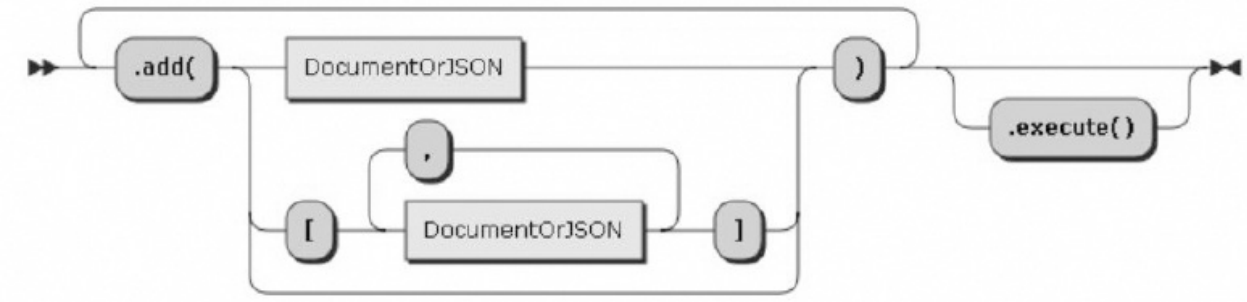
faire référence à un champ du même nom éventuellement présent dans la racine de chaque document JSON.
Les documents JSON seront stockés dans la colonne de nom `doc`.

Les fonctions utilisées dans la définition de la colonne virtuelle ont pour but d'extraire la valeur du champ `_id` qui doit être unique parmi tous les documents de la collection, sinon vous obtiendrez `ERROR (5116)`.

Ajouts

L'ajout d'un ou de plusieurs documents à une collection existante nécessite l'utilisation de la méthode `add`. Comme pour toutes les autres méthodes relatives aux collections, la méthode `execute` est implicitement exécutée en mode shell.

Figure 7-15 *Méthode d'ajout d'éléments à une collection*



Si aucun champ `_id` n'apparaît dans le premier niveau du document JSON, il est automatiquement ajouté et valué à l'aide d'un UUID. Si plusieurs documents doivent être ajoutés en une seule fois, il faudra les espacer à l'aide d'une virgule et utiliser les crochets ouvrants et fermants.

Tableau 7-68 Ajouts dans une collection JSON

En précisant l'identifiant	Avec plusieurs documents sans préciser l'identifiant

```
mysql-js>
db.tab_vol_json.add(
 { "_id" : "FE345_90",
   "num_vol" :"AF6143",
   "date_vol" :"2016-07-19",
   "desc_vol" :
   {"porte" : "20A",
    "plan_vol" : {
     "depart" : "21:45",
     "parking" : "3R2",
     "SID" : "AGN-2W",
     "STAR" : "TOU-5T",
     "arrivee" : null},
   "sieges" :
    [{"prenom" : "Guy",
      "nom" :    "Blanchet",
      "siege" :  "15A"},
     {"prenom" : "Gerard",
      "nom" :    "Diffis",
      "siege" :  "15B"},
     {"prenom" : "Victor",
      "nom" :    "Ferrage",
      "siege" :  "15C"}]
     }
  })
```

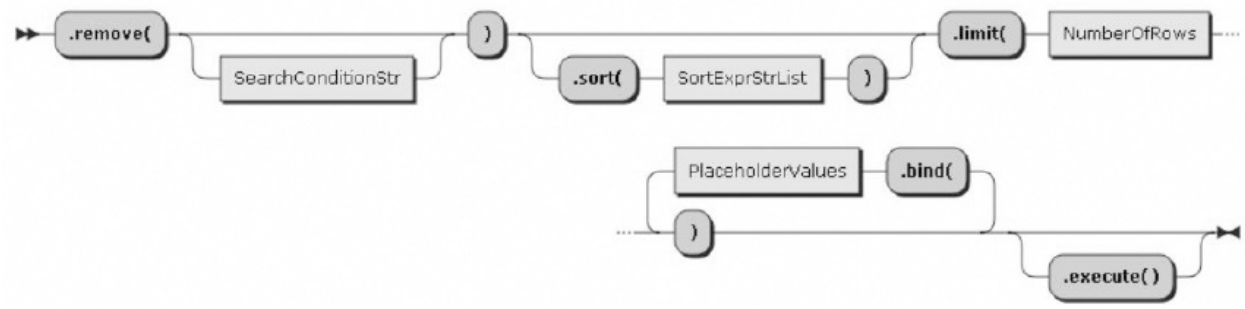
```
mysql-js>
db.tab_vol_json.add(
 [{ "num_vol" :"AF6143",
    "date_vol" :"2016-07-21",
    "desc_vol" : null }

 ,
  { "num_vol" :"AF6143",
    "date_vol" :"2016-07-22",
    "desc_vol" : null }
 ])
```

Suppressions

La suppression d'éléments d'une collection se réalise avec la méthode `remove`. Cette méthode peut cibler un élément en particulier à l'aide d'une condition (portant sur l'identifiant par exemple) ou une liste de documents triés sur un critère en utilisant conjointement les méthodes `sort` et `limit`.

Figure 7-16 *Méthode de suppressions d'éléments à une collection*



La première suppression concerne un seul document en particulier, la seconde concerne les trois premiers documents triés par le numéro de vol. Pour supprimer une collection dans son intégralité, il suffit de ne préciser aucun paramètre à l'appel de la méthode.

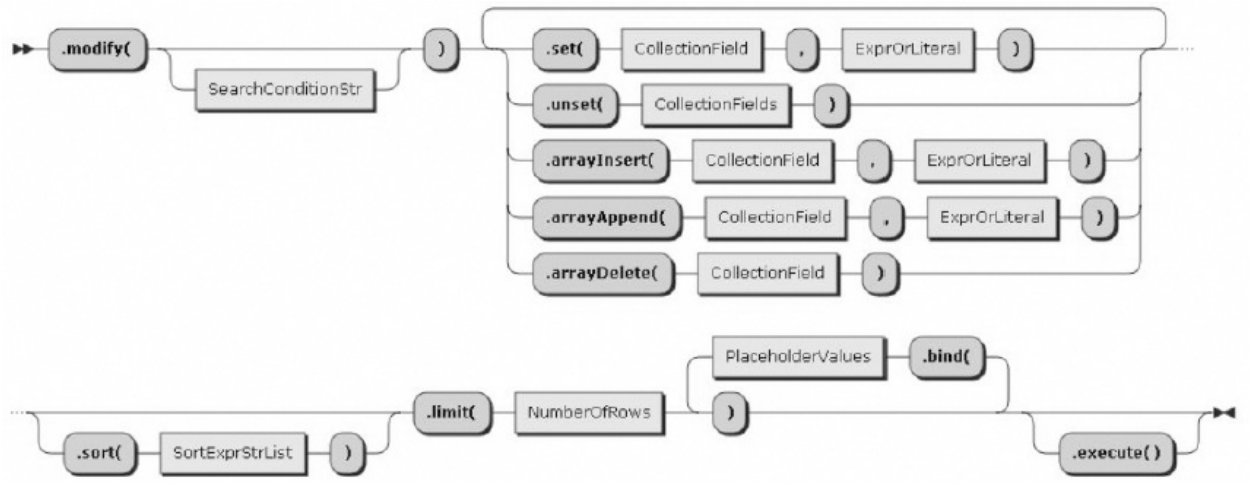
Tableau 7-69 Suppression de documents à une collection JSON

En précisant un critère	Sélection de plusieurs documents
`mysql-js> db.tab_vol_json.remove(` `"_id = 'FE345_90'")`	`mysql-js> db.tab_vol_json.remove().` `sort(["num_vol ASC"]).limit(3)`

Modifications

La modification d'éléments d'une collection se réalise avec la méthode `modify`. Cette méthode peut modifier un champ, qu'il soit atomique, composé (à l'aide de la notation pointée) ou de type tableau (vous pourrez ajouter, insérer ou supprimer un champ à un indice donné).

Figure 7-17 *Méthode de modification de champs dans une collection*



Le tableau suivant présente plusieurs modifications. Il semblerait qu'il ne soit pas possible de disposer `null` dans un champ à l'aide de cette méthode.

Tableau 7-70 Modification de champs dans des documents d'une collection JSON

Instructions	Commentaires

`mysql-js> db.tab_vol_json.`
`modify("_id = 'FE345_90'").`
`set("num_vol", "AF6147")`
	Modification d'un champ du premier niveau.

`mysql-js> db.tab_vol_json.`
`modify("_id = 'FE345_90'").`
`set("desc_vol.porte", "20G")`
	Modification d'une partie d'un champ composé.

`mysql-js> db.tab_vol_json.`
`modify("_id = 'FE345_90'").`
`set("desc_vol.plan_vol",`
`{"depart" : "21:55",`
`"parking" : "3R2",`
	Modification d'un champ

`"SID" : "AGN-2W",`
`"STAR" : "TOU-5T",`
`"arrivee" : "23:05"})`
	composé.

`mysql-js> db.tab_vol_json.`
`modify("_id = 'FE345_90'").`
`unset("desc_vol.plan_vol.parking")`
	Suppression d'une partie d'un champ composé.

`mysql-js> db.tab_vol_json.`
`modify("_id = 'FE345_90'").`
`arrayAppend("$.desc_vol.sieges",`
`{"prenom" : "Henri",`
`"nom" : "Fayat",`
`"siege" : "15D"})`
	Ajout d'un élément au tableau.

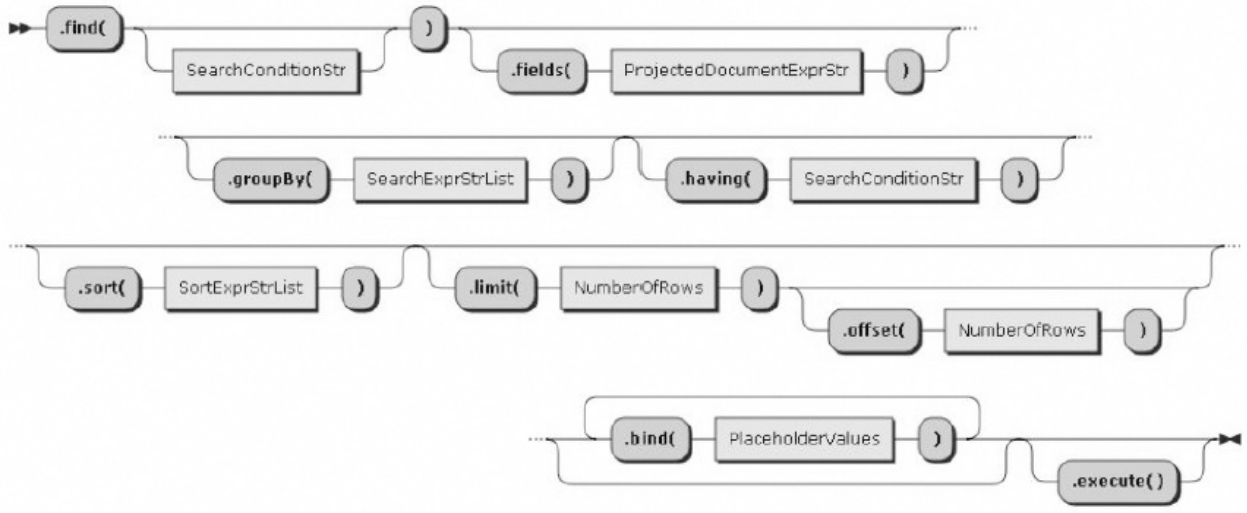
`mysql-js> db.tab_vol_json.`
`modify("_id = 'FE345_90'").`
`arrayDelete("$.desc_vol.sieges[0]")`
	Suppression du premier élément du tableau.

`mysql-js> db.tab_vol_json.`
`modify("_id = 'FE345_90'").`
`arrayInsert("$.desc_vol.sieges[0]",`
`{"prenom" : "Guy",`
`"nom" : "Blanchet",`
`"siege" : "15A"})`
	Ajout du premier élément au tableau.

Extractions

La recherche de documents sous critères est possible à l'aide de la méthode `find`. Cette méthode reprend les opérateurs du langage SQL : `OR`, `AND`, `XOR`, `IS`, `NOT`, `BETWEEN`, `IN`, `LIKE`, `!=`, `<>`, `>`, `>=`, `<`, `<=`, `&`, `|`, `<<`, `>>`, `+`, `-`, `*`, `/`, `~`, et `%`. Les méthodes `limit`, `sort` et `skip` vous permettront de limiter et trier vos résultats. Pour restituer une collection dans son intégralité, il suffit de ne préciser aucun paramètre à l'appel de la méthode.

Figure 7-18 *Méthode d'extraction des collections*



Le tableau suivant présente plusieurs extractions. La troisième illustre la nécessité de la notation pointée pour accéder au champ `porte`. La quatrième présente le passage d'un paramètre avec `bind`. La cinquième introduit la méthode `fields` pour se restreindre à des champs en particulier. L'accès à des éléments d'un tableau s'opère à l'aide des crochets. La dernière extraction trie des détails des documents du vol AF6143, le résultat est un tableau.

Tableau 7-71 Extraction de champs dans des documents d'une collection JSON

Instructions	Résultats
`mysql-js> db.tab_vol_json.` `  **find**("_id = 'FE345_90'")`	Document numéro FE345_90 entier.

`mysql-js> db.tab_vol_json.`
`  **find**("num_vol = 'AF6143'`
`  AND  date_vol = '2016-07-19'")`
	Documents du vol AF6143 à la date du 19/7/2016.

`mysql-js> db.tab_vol_json.`
`  find("num_vol = 'AF6143' AND`
`  desc_vol.porte='20B'")`
	Documents correspondants au numéro de vol AF6143 et en porte 20B.

`mysql-js> db.tab_vol_json.`
`  **find**("date_vol = :dvol").`
`  bind("dvol", "2016-07-19")`
	Documents à la date du 19/7/2016.

`mysql-js> db.tab_vol_json.`
`  **find**("date_vol = :dvol").`
`  fields(["desc_vol.porte",`
`  "desc_vol.plan_vol.depart"]).`
`  bind("dvol", "2016-07-19")`
	Numéros de porte des documents à la date du 19/7/2016.

`mysql-js> db.tab_vol_json.`
`  **find**("date_vol = :dvol").`
`  fields(["desc_vol.sieges[0].prenom",`
`  "desc_vol.sieges[0].nom"]).`
`  bind("dvol", "2016-07-19")`
	Prénom et nom des passagers du 19/7/2016.

`mysql-js> db.tab_vol_json.`
`  **find**("num_vol = 'AF6143'").`
`  fields(["date_vol",`
`  "desc_vol.porte",`
`  "desc_vol.plan_vol.depart"]).`
`  sort(["date_vol ASC"]).limit(2)`
	`[ {"date_vol": "2016-07-19",`
	`  "desc_vol.plan_vol.depart": "21:45",`
	`  "desc_vol.porte": "20A"`
	`  },`
	`  {"date_vol": "2016-07-20",`
	`  "desc_vol.plan_vol.depart": "21:40",`
	`  "desc_vol.porte": "20B"`
	`  } ]`
	`2 documents in set (0.00 sec)`

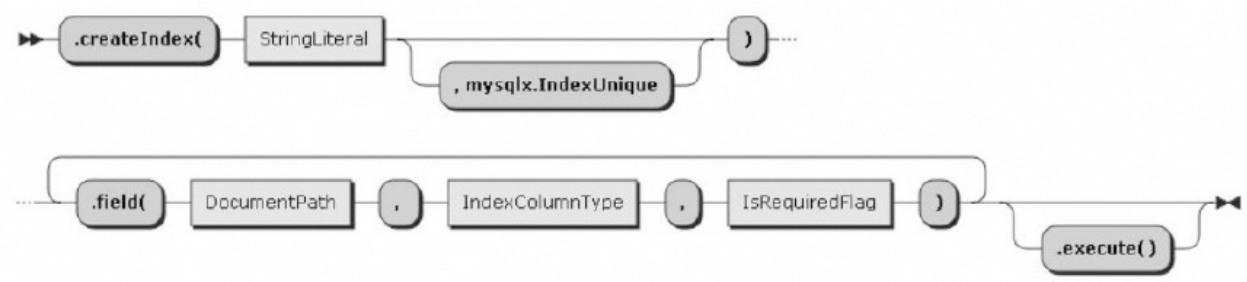
Indexation

Mis à part le champ `_id` qui joue le rôle d'une clé primaire et qui est par conséquent déjà indexé, il est possible de définir des index (uniques ou non) sur les autres champs d'un document. Il n'est pas possible de définir un index

sur une composition de champs.

La syntaxe suivante prévoit de positionner le chemin du champ à indexer de même que le type (`INTEGER`, `TEXT`...) dans la méthode `field`. Le dernier paramètre autorise ou non l'absence de valeur dans le champ (`NULL`). Pour l'ajout comme la suppression d'un index, la méthode `execute` doit être explicitement citée.

Figure 7-19 *Méthode de création d'un index*



L'instruction suivante déclare un index non unique sur le champ `porte` et permet à certains documents de ne pas renseigner cette donnée. La méthode `dropIndex` ne nécessitera que le nom de l'index pour supprimer ce dernier.

```
db.tab_vol_json.createIndex("idx_porte").
                field("desc_vol.porte", "TEXT(3)", false).execute()
```

Méthodes pour les documents JSON

Il est possible d'utiliser ce nouveau shell pour manipuler des tables conventionnelles contenant des documents JSON (par l'intermédiaire du type `JSON` présent depuis la version 5.7.8). La méthode `getTables` liste les tables présentes dans le schéma (ou la database) connecté(e). D'autres méthodes préfixées par `db.nom_table` (`insert`, `select`, `update` et `delete`) visent à manipuler les tables à la façon CRUD. Il est inutile de valider par `commit` ces opérations, le shell s'en charge. Le terme `db` désigne la base sélectionnée dans le schell (`\use` *`nom_base`*). Considérons la table suivante.

Tableau 7-72 Stockage de documents JSON

Création de la table	Commentaires

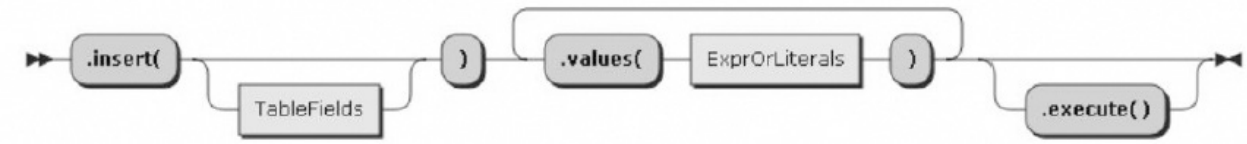
`CREATE TABLE bd_json.tab_vol_json2`
`(id_vol INTEGER AUTO_INCREMENT`
`PRIMARY KEY,`
	La première colonne est la clé primaire, la deuxième permettra de stocker des

`doc JSON,`
`modif DATETIME`
`) ENGINE=InnoDB DEFAULT CHARSET=utf8;`
	documents JSON et la dernière est la date de dernière modification.

Insertions

La méthode `insert` permet d'insérer une ou plusieurs lignes dans une table. La méthode `values` liste les valeurs à substituer aux colonnes citées dans `insert`. Comme dans les exemples qui suivent, vous devrez enlever toute indentation à vos contenus JSON. Par ailleurs, ne comptez pas utiliser des fonctions SQL (`SYSDATE`, `CURRENT_TIME`...) dans la méthode `values`, le shell ne le permet pas pour l'instant.

Figure 7-20 *Méthode d'insertion de lignes*



Le tableau suivant présente deux insertions.

Tableau 7-73 Insertions de lignes via MySQL Shell

Instructions	Commentaires

`mysql-js> db.tab_vol_json2.insert`
`("doc","modif").`
`values(`
`'{"date_vol":"`
`values('`
`{"date_vol":"AF6143","num_vol":"2016-09-09",`
`"desc_vol":{"porte":"20A", "plan_vol":`
`{"depart": "21:45", "parking":"13R",`
`"SID": "AGN-2S","STAR":"TOU-5T",`
`"arrivee":null},`
`sieges":[{"prenom":"Guy","nom":"Blanchet",`
`"siege":"10A"},`
`{"prenom":"Gerard","nom":"Diffis",`
`"siege":"10B"},`
`{"prenom":"Victor","nom":"Ferrage",`
`"siege":"10C"}]}}',`
`'2016-09-0919:30:00')`
	Deux colonnes sont renseignées, la première (`id_vol`) est autoincrémentée. Tous les champs sont renseignés à part un (`arrivee`), le tableau (`sieges`) contient 3 passagers.

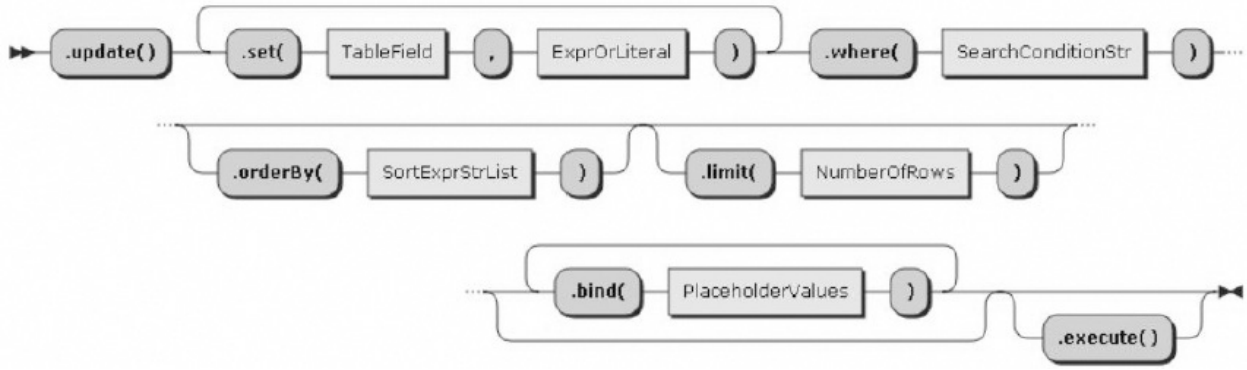
`mysql-js> db.tab_vol_json2.insert`
`("doc","modif").`
`values('`
`{"date_vol":"AF6144","num_vol":"2016-09-09",`
`"desc_vol":{"porte":"20B",`
`"plan_vol":{"depart": "22:15",`
`"parking":"13R","SID": "AGN-2S",`
`"STAR":"TOU-5T","arrivee":null},
	Deux colonnes sont renseignées, la première (`id_vol`) est autoincrémentée. Le tableau (`sieges`) et le

`"sieges":null}}',` `'2016-09-09 19:50:00')`	champ `arrivee` ne sont pas renseignés.

Modifications

La méthode `update` permet de modifier une ou plusieurs lignes dans une table. Le filtre des lignes à modifier s'opère dans la méthode `where` en sélectionnant éventuellement à l'aide d'un tri (méthode `orderBy`) et en limitant les résultats (méthode `limit`). Comme dans la méthode `find` qui recherche dans les collections, la méthode `bind` sert à passer des paramètres par valeur au lieu de les inclure en constantes dans la condition de recherche.

Figure 7-21 *Méthode de modification de lignes*



Vous devrez enlever toute indentation à vos contenus JSON dans la méthode `set` et l'utilisation de fonctions SQL n'est pas possible. Par ailleurs, il est impossible d'accéder au détail d'un contenu JSON (en revanche la totalité du document peut être modifiée). Seul le nom des colonnes de la table est accepté dans la méthode `set`. Le tableau suivant présente quelques modifications.

Tableau 7-74 Modification de lignes via MySQL Shell

Instructions	Commentaires

`mysql-js> db.tab_vol_json2.update().`
`set("doc", "null").`
`where("id_vol=3")`
	Supression du contenu JSON du troisième vol.

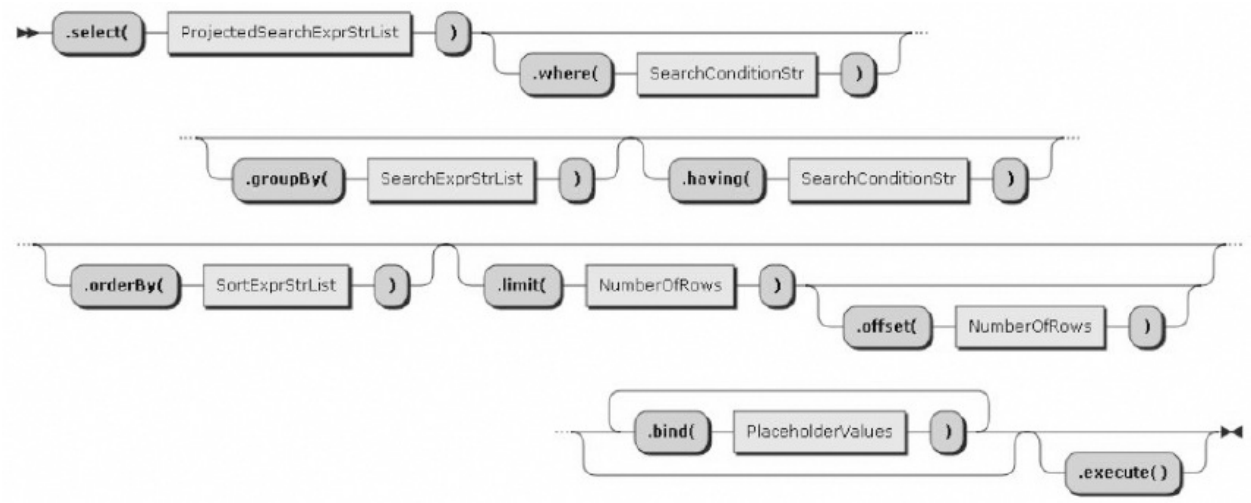
`mysql-js> db.tab_vol_json2.update().`
`set("modif", "2016-09-09 19:35:00").`
`where("id_vol=1")`
	Mise à jour de la colonne `modif` du premier vol.

`mysql-js> db.tab_vol_json2.update().`
`set("doc", '{"num_vol":"AF6148",`
`"date_vol":"2016-09-10",`
`"desc_vol":null}').`
`where("id_vol=3")`
	Mise à jour du contenu JSON du troisième vol.

Extractions

La méthode `select` sert à rechercher des lignes sous critères. La méthode `select` reprend les opérateurs de SQL, à savoir `OR`, `AND`, `XOR`, `IS`, `NOT`, `BETWEEN`, `IN`, `LIKE`, `!=`, `<>`, `>`, `>=`, `<`, `<=`, `&`, `|`, `<<`, `>>`, `+`, `-`, `*`, `/`, `~`, et `%`. Comme dans la syntaxe du langage SQL, on retrouve les méthodes `where`, `groupBy`, `having`, `orderBy` et `limit` pour limiter et trier les résultats. La méthode `bind` permettra de paramétrer les conditions.

***Figure 7-22** Méthode de sélection de lignes*



Cette méthode n'est pas adaptée à l'analyse des contenus JSON comme `find` peut le proposer. Avec `select`, il n'est possible que d'extraire un document dans son intégralité. Le tableau suivant présente quelques extractions. La première concerne les deux premières colonnes de la table. La deuxième utilise deux paramètres et restitue un document. La dernière opère un tri et limite à deux lignes dans le résultat. Pour restituer une table dans son intégralité il suffit de ne préciser aucun paramètre à l'appel de la méthode.

Tableau 7-75 Extractions de lignes via MySQL Shell

Instructions	Résultats

`mysql-js> db.tab_vol_json2.`
`  **select**(`
`  ["id_vol","modif"])`
	`+--------+---------------------+`
	`| id_vol | modif |`
	`+--------+---------------------+`
	`| 1 | 2016-10-09 19:35:00 |`
	`| 2 | 2016-10-09 19:50:00 |`
	`| 3 | 2016-10-09 20:00:00 |`
	`+--------+---------------------+`
	`3 rows in set (0.02 sec)`

```
mysql-js> db.tab_vol_
json2.select(["doc"]).
  where("id_vol=:idvol
        AND
        modif=:dvol").
      bind("idvol",1).
      bind("dvol",
      "2016-09-09 19:35:00")
```

```
| doc
+-------------------------------------------
| {"num_vol": "AF6143", "date_vol": "2016-09-09",
"desc_vol": {"porte": "20A", "sieges": [{"nom":
"Blanchet", "siege": "10A", "prenom": "Guy"},
{"nom":
"Diffis", "siege": "10B", "prenom": "Gerard"},
{"nom": "Ferrage", "siege": "10C", "prenom":
"Victor"}], "plan_vol": {"SID": "AGN-2S", "STAR":
"TOU-5T", "depart": "21:45", "arrivee": null,
"parking": "13R"}}} |
-------------------------------------------+
1 row in set (0.00 sec)
```

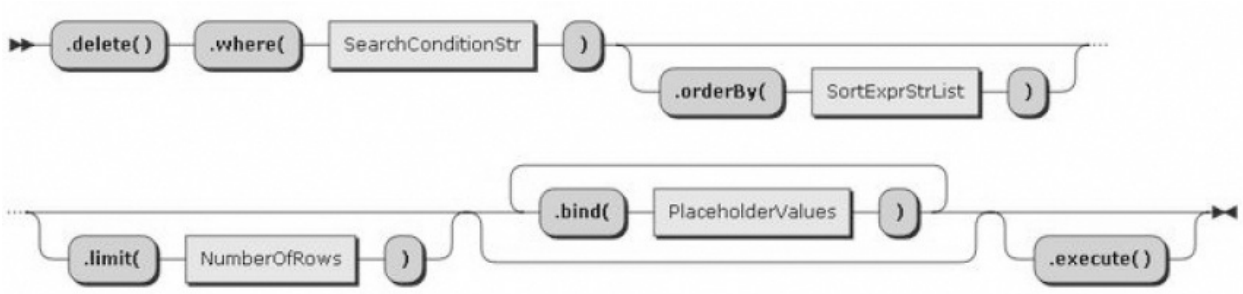
```
mysql-js> db.tab_vol_json2.
   select(["id_vol","modif"]).
   orderBy(["modif desc"]).
   limit(2)
```

```
+--------+--------------------+
| id_vol | modif              |
+--------+--------------------+
|      3 | 2016-10-09 20:00:00 |
|      2 | 2016-10-09 19:50:00 |
+--------+--------------------+
2 rows in set (0.00 sec)
```

Suppressions

La méthode `delete` supprime des lignes sous critères précisés à l'aide de la méthode `where` (identique à celle qu'on trouve dans la méthode `select`). Les méthodes `bind` et `limit` ont les mêmes fonctionnalités que pour d'autres méthodes précédemment décrites.

Figure 7-23 *Méthode de suppression de lignes*



Il n'est pas possible de supprimer des lignes en testant un fragment de document JSON. Le seul moyen est de comparer ce document dans son intégralité. L'instruction suivante supprime les lignes de la table qui vérifient la condition. Pour supprimer une table dans son intégralité, il ne faut préciser aucun paramètre à la méthode `where`.

```
db.tab_vol_json2.delete().
       where("id_vol = 1 AND modif = '2016-09-09 19:35:00'")
```

Fonctions SQL pour JSON

Présent depuis la version 5.7.8, le type de données `JSON` est dédié à la gestion de documents JSON. Ce type de colonnes ne peut pas avoir de valeur par défaut et la taille maximale d'un contenu est régie par le paramètre `max_allowed_packet` (de 4 Mo à 1 Go selon les configurations). Comme d'autres types binaires, le type JSON ne permet pas une indexation directe mais seulement par l'intermédiaire de colonnes virtuelles (voir plus loin). Concernant SQL, des nouvelles fonctions sont apparues pour :

- générer du contenu (`JSON_ARRAY`, `JSON_OBJECT`, `JSON_QUOTE et JSON_OBJECTAGG`) ;
- extraire du contenu (`JSON_CONTAINS`, `JSON_CONTAINS_PATH`, `JSON_EXTRACT` et `JSON_KEYS`) ;
- modifier du contenu (`JSON_UNQUOTE`, `JSON_ARRAY_APPEND`, `JSON_ARRAY_INSERT`, `JSON_INSERT`, `JSON_MERGE`, `JSON_REMOVE`, `JSON_REPLACE` et `JSON_ARRAYAGG`) ;
- extraire certaines caractéristiques du contenu (`JSON_DEPTH`, `JSON_LENGTH`, `JSON_TYPE` et `JSON_VALID`).

Création de la table

Une fois la table créée, tout document est validé dès l'insertion et après chaque modification. Ainsi, il n'est pas possible de stocker un document invalide (*`ERROR 3140 (22032)`*:..). Dans l'exemple suivant, il manque une double quote à la donnée du numéro de vol. Cette chaîne de caractères, qui n'est pas valide d'un point de vue JSON, peut être insérée dans la colonne `doc_chaine`.

Tableau 7-76 Création d'une table et tentative d'insertion

Création de la table	Résultats

`CREATE TABLE tab_vol_json3`
`(id SMALLINT AUTO_INCREMENT`
`PRIMARY KEY,`
`utilisateur VARCHAR(30),`
`doc` **`JSON`**`,`
`doc_chaine VARCHAR(500),`
`modif DATETIME`
`) ENGINE=InnoDB DEFAULT CHARSET=utf8;`
	`mysql> INSERT INTO tab_vol_json3`
	`-> (utilisateur, doc, modif) VALUES`
	`-> ('Brouard',`
	`-> '{`**`"date_vol":"2016-07-18",`**
	`"num_vol":AF6140``}', SYSDATE());`
	`ERROR 3140 (22032): Invalid JSON text:`
	`"Invalid value." at position 35 in value`
	`for column 'tab_vol_json.doc'.`

Insertion de documents

Le tableau suivant présente des insertions valides. La structure du document JSON inclut des champs, sous-champs et un tableau (*`sieges`*). Deux autres lignes sont ajoutées à la table pour fournir des résultats aux requêtes par la suite.

Tableau 7-77 Insertions de documents

Instructions	Commentaires
`INSERT INTO tab_vol_json3 (utilisateur, doc_chaine, modif) VALUES ('Brouard', '{"date_vol":"2016-07-18", "num_vol":AF6140"}', SYSDATE());`	Insertion d'un document JSON valide dans une colonne VARCHAR.
`INSERT INTO tab_vol_json3 (utilisateur, doc, modif) VALUES ('Brouard', '{"date_vol":"2016-07-18", "num_vol":"AF6141"}', SYSDATE());`	Insertion d'un document JSON valide dans une colonne JSON.
`INSERT INTO tab_vol_json3 (utilisateur, doc, modif) VALUES ('Salais', '{"_id" : "FE345_90", "num_vol" : "AF6143", "date_vol" : "2016-07-19", "sequence" : 345, "desc_vol" : {"porte" : "20A", "plan_vol" : { "depart" : "21:45", "parking" : "3R2"}, "sieges" : [{"prenom" : "Guy", "nom" : "Blanchet", "siege" : "15A"}, {"prenom" : "Gerard", "nom" : "Diffis", "siege" : "15B"}, {"prenom" : "Victor", "nom" : "Ferrage", "siege" : "15C"}, {"prenom" : "Henri", "nom" : "Alquie", "siege" : "13C"}] } }', DATE_ADD(SYSDATE(),INTERVAL 1 DAY));`	Insertion d'un document JSON structuré composé de cinq champs au premier niveau. Le quatrième de ces champs (`desc_vol`) est lui-même composé de trois champs dont un tableau.

Générer du contenu

Le tableau suivant présente quelques contenus JSON générés. La fonction `JSON_ARRAY ([expr[, expr]...])` construit un tableau. La fonction `JSON_OBJECT([champ, expr[, champ, expr]...])` construit un document avec des champs. La fonction `JSON_QUOTE(doc_json)` encadre un contenu entre doubles quotes en préservant éventuellement des quotes interieures.

Tableau 7-78 Génération de contenu JSON

Instructions	Résultat

`SELECT **JSON_ARRAY**(id, utilisateur, modif) FROM tab_vol_json3 WHERE id BETWEEN 2 AND 4;`	`+-----------------------------------------+`
	`| JSON_ARRAY(id, utilisateur, modif) |`
	`+-----------------------------------------+`
	`| [2, "Brouard", "2016-09-24 12:26:22.000000"] |`
	`| [3, "Salais", "2016-09-25 12:26:22.000000"] |`
	`| [4, "Bizoi", "2016-09-25 12:26:22.000000"] |`
	`+-----------------------------------------+`

`SELECT **JSON_OBJECT**('num_v', id, 'compte', JSON_ARRAY(utilisateur, DATE(modif))) FROM tab_vol_json3 WHERE utilisateur <> 'Brouard';`	`+-----------------------------------------+`
	`| {"num_v": 3, "compte": ["Salais", "2016-09-25"]}`
	`| {"num_v": 4, "compte": ["Bizoi", "2016-09-25"]}`
	`| {"num_v": 5, "compte": ["Salais", "2016-09-26"]}`
	`+-----------------------------------  ------+`

`SELECT id, **JSON_QUOTE**(utilisateur) FROM tab_vol_json3 WHERE id IN (1,3,4);`	`+----+------------------------+`
	`| id | JSON_QUOTE(utilisateur) |`
	`+----+------------------------+`
	`|  1 | "Brouard"              |`
	`|  3 | "Salais"               |`
	`|  4 | "Bizoi"                |`
	`+----+------------------------+`

Extraire du contenu

Les tableaux suivants présentent les fonctions pour parcourir et extraire du contenu JSON. La fonction `JSON_CONTAINS(doc_json, expr[, chemin])` retourne `1` si l'expression est trouvée dans le chemin à l'intérieur du document, `0` sinon. Notez l'utilisation des doubles quotes pour comparer une chaîne de caractères et le caractère `$` qui désigne la racine. La première requête ne trouve pas la valeur `348` pour le champ *`sequence`*. La deuxième requête extrait les lignes qui vérifient que le champ *`porte`* soit égal à `20A` (utilisez la notation pointée pour parcourir la structure) ou que le champ *`parking`* n'est pas renseigné.

La fonction `JSON_CONTAINS_PATH(doc_json, {'one'|'all'}, chemin[, chemin]...)` retourne `1` si le (ou les) chemin(s) existe(nt) à l'intérieur du document, `0` sinon. un document avec des champs. La troisième requête cherche les documents ne disposant pas du champ *`desc_vol`*. La dernière s'intéresse à ceux qui disposent d'un champ *`porte`* (il n'est pas dit qu'il ne puisse pas être `null`), et d'un quatrième passager.

Tableau 7-79 Parcours de contenu JSON

Instructions	Résultat

`SELECT`
` **JSON_CONTAINS**(doc, '"AF6143"' ,`
`  '$.num_vol')`
`  AS contains_AF6143,`
` **JSON_CONTAINS**(doc, '348' ,'$.sequence')`
	`+-----------------+------------------+`
	`| contains_AF6143 | contains_seq_348 |`
	`+-----------------+------------------+`

```
               AS contains_seq_348          |               1 |                 0 |
FROM   tab_vol_json3                        +-----------------+-------------------+
WHERE  utilisateur = 'Salais'
AND    id = 3;
```

```
SELECT id, utilisateur, DATE(modif)
FROM   tab_vol_json3
WHERE  JSON_CONTAINS(doc, '"20A"' ,
       '$.desc_vol.porte') = 1
OR     JSON_CONTAINS(doc, 'null' ,
       '$.desc_vol.plan_vol.parking')=1
ORDER BY id;
```

```
+----+-------------+-------------+
| id | utilisateur | DATE(modif) |
+----+-------------+-------------+
|  3 | Salais      | 2016-09-25  |
|  4 | Bizoi       | 2016-09-25  |
|  5 | Salais      | 2016-09-26  |
+----+-------------+-------------+
```

```
SELECT id, utilisateur, DATE(modif)
FROM   tab_vol_json3
WHERE  JSON_CONTAINS_PATH(doc, 'one' ,
                          '$.desc_vol')=0;
```

```
+----+-------------+-------------+
| id | utilisateur | DATE(modif) |
+----+-------------+-------------+
|  2 | Brouard     | 2016-09-24  |
+----+-------------+-------------+
```

```
SELECT id, utilisateur, DATE(modif)
FROM   tab_vol_json3
WHERE  JSON_CONTAINS_PATH(doc, 'all' ,
        '$.desc_vol.porte',
        '$.desc_vol.sieges[3].siege')=1;
```

```
+----+-------------+-------------+
| id | utilisateur | DATE(modif) |
+----+-------------+-------------+
|  3 | Salais      | 2016-09-25  |
+----+-------------+-------------+
```

La fonction `JSON_EXTRACT(doc_json, chemin [,chemin] ...)` extrait le contenu désigné dans le (ou les) chemin(s). Depuis la version 5.7.9, l'opérateur `->` joue le même rôle si seulement un chemin est décrit. Les deux premières requêtes extraient le numéro et la date du vol. La première parcourt tous les éléments du tableau *`sieges`* tandis que la deuxième extrait le champ *`porte`* sous le champ *`desc_vol`*.

La fonction `JSON_KEYS(`*`doc_json`* `[,` *`chemin`*`])` liste les champs, sous la forme d'un tableau, qui sont présents au niveau du chemin précisé. La dernière requête permet de constater que la première ligne est associée à un document absent. Le deuxième document ne dispose que des champs concernant le numéro et la date du vol. Enfin le troisième document est le plus complet en termes de structure.

Tableau 7-80 Extraction de contenu JSON

Instructions	Résultat

`SELECT`
` JSON_EXTRACT(doc,`
`  '$.num_vol', '$.date_vol') AS elements_vol,`
` JSON_EXTRACT(doc,`
`  '$.desc_vol.sieges[*].siege')`
`  AS passagers`
`FROM  tab_vol_json3`
`WHERE  utilisateur='Salais';`
	`+-------------------------+-----------------------+`
	`| elements_vol | passagers |`
	`+-------------------------+-----------------------+`
	`| ["AF6143", "2016-07-19"] | ["15A", "15B", "15C",`
	`"13C"]`
	`| ["AF6143", "2016-07-20"] | ["1A", "1B", "1C"]`
	`+-------------------------+-----------------------+`

```
SELECT id,
```

```
 CONCAT(doc->'$.num_vol',
        doc->'$.date_vol')
     AS elt_vol,
 doc->'$.desc_vol.porte'
     AS porte
FROM   tab_vol_json3
WHERE  utilisateur='Salais';
```

```
+----+---------------------+-------+
| id | elt_vol             | porte |
+----+---------------------+-------+
|  3 | "AF6143""2016-07-19" | "20A" |
|  5 | "AF6143""2016-07-20" | "20B" |
+----+---------------------+-------+
```

```
SELECT id,
 JSON_KEYS(doc) AS sous_doc,
 JSON_KEYS(doc,
         '$.desc_vol.plan_vol')
 AS sous_plan_vol
FROM   tab_vol_json3
WHERE  id IN (1,2,4);
```

```
+----+-------------------------+-------------------+
| id | sous_doc                | sous_plan_vol
+----+-------------------------+-------------------+
|  1 | NULL                    | NULL
|  2 | ["num_vol", "date_vol"] | NULL
|  4 | ["_id", "num_vol",
       "date_vol", "desc_vol",
       "sequence"]             |
["depart","parking"]
+----+-------------------------+--------------------
+
```

Retyper du contenu

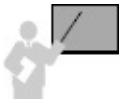
La fonction `JSON_UNQUOTE(doc_json)` supprime les doubles quotes qui encadrent le contenu. En combinant cette fonction avec `CAST`, il est possible de choisir le type de données résultat avant un traitement éventuel (dans l'exemple suivant, une chaîne, une date et un numérique).

Tableau 7-81 Retyper du contenu JSON

Instructions	Résultat

`SELECT id,`
` JSON_UNQUOTE(doc->'$.num_vol') AS num_vol,`
` JSON_UNQUOTE(doc->'$.date_vol') AS date_vol,`
` JSON_UNQUOTE(doc->'$.desc_vol.sieges[0].siege')`
`  AS siege_0`
`FROM  tab_vol_json3`
`WHERE  utilisateur = 'Salais';`
	`+----+---------+------------+---------+`
	`| id | num_vol | date_vol  | siege_0 |`
	`+----+---------+------------+---------+`
	`|  3 | AF6143  | 2016-07-19 | 15A     |`
	`|  5 | AF6143  | 2016-07-20 | 1A      |`
	`+----+---------+------------+---------+`

`SELECT id,`
` CAST(JSON_UNQUOTE(doc->'$.num_vol')`
`  AS CHAR(6)) AS num_vol,`
` CAST(JSON_UNQUOTE(doc->'$.date_vol')`
`  AS DATE) AS date_vol,`
` CAST(JSON_UNQUOTE(doc->'$.sequence')`
`  AS UNSIGNED INTEGER) AS sequence`
`FROM  tab_vol_json3`
`WHERE  utilisateur = 'Salais';`
	`+----+---------+------------+----------+`
	`| id | num_vol | date_vol  | sequence |`
	`+----+---------+------------+----------+`
	`|  3 | AF6143  | 2016-07-19 |      345 |`
	`|  5 | AF6143  | 2016-07-20 |      347 |`
	`+----+---------+------------+----------+`



Depuis la version 8, l'opérateur `->>` enrichit l'opérateur `->` (qui permettait d'extraire un contenu désigné) en supprimant les quotes du résultat extrait. Cet opérateur peut apparaître dans `SELECT`, `WHERE`, `GROUP BY`, `HAVING` et `ORDER BY`. Ainsi, les trois écritures sont équivalentes :

- `JSON_UNQUOTE(JSON_EXTRACT(`*`expr`*`,`*`chemin`*`))`
- `JSON_UNQUOTE(`*`expr`*`->`*`chemin`*`)`
- *`expr`*`->>`*`chemin`*

La requête précédente se simplifie de la manière suivante :

```
SELECT id, CAST(doc->>'$.num_vol' AS CHAR(6)) AS num_vol,
           CAST(doc->>'$.date_vol' AS DATE) AS date_vol,
            CAST(doc->>'$.sequence' AS UNSIGNED INTEGER) AS sequence
FROM   bd_json.tab_vol_json3
WHERE  utilisateur = 'Salais';
```

Modifier du contenu

Concernant la mise à jour de contenus, plusieurs fonctions existent. La fonction `JSON_INSERT(`*`doc_json`*`, `*`chemin`*`, `*`expr`*`[,`*`chemin`*`, `*`expr`*`]...)` permet d'insérer des champs à différents niveaux du contenu qui passe en paramètre. La fonction `JSON_ARRAY_APPEND(`*`doc_json`*`, `*`chemin`*`, `*`expr`*`[,`*`chemin`*`, `*`expr`*`]...)` ajoute un élément après le dernier indice d'un tableau (en précisant éventuellement un niveau si des tableaux sont imbriqués). Dotée des mêmes paramètres, la fonction `JSON_ARRAY_INSERT` ajoute un élément à un indice choisi d'un tableau.

La première requête ajoute deux champs (*`sequence`* et *`desc_vol`* lui-même composé) à un contenu initial composé de deux champs (numéro et date du vol). La deuxième requête ajoute un passager au tableau *`sieges`* du vol numéro 3. Pour remplacer effectivement un fragment dans la table, il suffit d'utiliser `UPDATE` (voir plus loin). La dernière requête ajoute le même passager au premier indice du tableau.

Tableau 7-82 Modifier du contenu JSON

Instructions	Résultat
`SELECT` **`JSON_INSERT`**`(doc,`	

'$.sequence',344, '$.desc_vol', JSON_OBJECT('porte', '14K', 'plan_vol', JSON_OBJECT('depart', null, 'parking','3R4'))) FROM tab_vol_json3 WHERE id = 2;	{"num_vol": "AF6141", "date_vol": "2016-07-18", **"desc_vol": {"porte": "14K", "plan_vol": {"depart": null, "parking": "3R4"}}**, **"sequence": 344**}
SELECT **JSON_ARRAY_APPEND**(JSON_EXTRACT(doc,'$.desc_vol.sieges'), '$', JSON_OBJECT('prenom','Antoine', 'nom','Derouin', 'siege', '5A')) FROM tab_vol_json3 WHERE id = 3;	[{"nom": "Blanchet", "siege": "15A", "prenom": "Guy"}, {"nom": "Diffis", "siege": "15B", "prenom": "Gerard"}, {"nom": "Ferrage", "siege": "15C", "prenom": "Victor"}, {"nom": "Alquie", "siege": "13C", "prenom": "Henri"}, **{"nom": "Derouin", "siege": "5A", "prenom": "Antoine"}**]
SELECT **JSON_ARRAY_INSERT**(JSON_EXTRACT(doc,'$.desc_vol.sieges'), '$[0]', JSON_OBJECT('prenom','Antoine', 'nom','Derouin', 'siege', '5A')) FROM tab_vol_json3 WHERE id = 3;	[**{"nom": "Derouin", "siege": "5A", "prenom": "Antoine"}**, {"nom": "Blanchet", "siege": "15A", "prenom": "Guy"}, {"nom": "Diffis", "siege": "15B", "prenom": "Gerard"}, {"nom": "Ferrage", "siege": "15C", "prenom": "Victor"}, {"nom": "Alquie", "siege": "13C", "prenom": "Henri"}] |

Pour fusionner plusieurs contenus, il existe la fonction JSON_MERGE(*doc_json*, [,*doc_json*]...). Pour enlever un fragment de contenu, il faudra utiliser la fonction JSON_REMOVE(*doc_json*, *chemin* [,*chemin*]...).

Par ailleurs, les fonctions JSON_SET(*doc_json*, *chemin*, *expr*[,*chemin*, *expr*]...) et JSON_REPLACE ayant les mêmes paramètres remplacent des fragments de contenus. La méthode JSON_SET a la capacité d'ajouter de nouveaux fragments sans qu'ils existent au préalable, alors que JSON_INSERT insère un fragment sans remplacer aucun champ existant.

La première requête du tableau suivant fusionne deux contenus alors que la deuxième supprime les champs *desc_vol* et *sieges* du document. La troisième requête modifie deux champs et en ajoute un au document. La dernière remplace trois champs d'un document.

Tableau 7-83 Modifier du contenu JSON

Instructions	Résultat
SELECT **JSON_MERGE**((JSON_OBJECT('num_v', id)), doc) FROM tab_vol_json3 WHERE id = 2;	+----------+ | merge | +----------+ | **{"num_v": 2**, "num_vol": "AF6141", "date_vol": "2016-07-18"} | +----------+
SELECT **JSON_REMOVE**(doc, '$.sieges','$.desc_vol') FROM tab_vol_json3 WHERE id = 2;	+----------+ | {"num_vol": "AF6141", "date_vol": "2016-07-18"} | +----------+

```
SELECT JSON_SET(doc,
       '$._id', 'FG00_02',
       '$.date_vol', '2016-07-23',
       '$.num_vol' , 'AF5661')
FROM   tab_vol_json3
WHERE  id = 2;
```

```
+--------------------------------------+
| {"_id": "FG00_02", "num_vol":
"AF5661",
   "date_vol": "2016-07-23"}
                                       |
+--------------------------------------+
```

```
SELECT JSON_REPLACE(doc,
  '$._id', 'FG450_92',
  '$.desc_vol.porte', '20F',
  '$.desc_vol.sieges[0].siege', '12G')
FROM   tab_vol_json3
WHERE  utilisateur = 'Bizoi';
```

```
+--------------------------------------+
| {"_id": "FG450_92", "num_vol":
"AF6146",
"date_vol": "2016-07-19", "desc_vol":
{"porte": "20F", "sieges": [{"nom":
"Blanchet", "siege": "12G", "prenom":
"Guy"},
{"nom": "Diffis", "siege": "3B",
"prenom":
"Gerard"}], "plan_vol": {"depart":
"21:55",
"parking": "3R6"}}, "sequence":
346}
|
+--------------------------------------+
```

Extraire des caractéristiques

Il existe des fonctions qui renseignent sur les caractéristiques des documents JSON stockés. La fonction `JSON_DEPTH(`*`doc_json`*`)` retourne la « profondeur » maximale du document (1 pour un scalaire, 2 pour un objet qui contient un champ ou pour un tableau qui contient un scalaire, etc.). De même, la fonction `JSON_LENGTH(`*`doc_json`*`[, `*`chemin`*`])` retourne la taille du document en considérant d'abord le nombre d'objets membres et en considérant la taille d'un tableau comme le nombre d'éléments de celui-ci.

La première requête du tableau suivant renseigne sur les niveaux de la structure des documents sélectionnés. La deuxième requête présente deux tailles : celle du document dans sa globalité (nombre de champs du premier niveau) et celle du tableau *sieges* (nombre d'éléments dans chaque collection).

Tableau 7-84 Renseigner sur du contenu JSON

Instructions	Résultat

`SELECT id, utilisateur,`
`  JSON_DEPTH(doc) AS profondeur_maxi`
`FROM  tab_vol_json3`
`WHERE  utilisateur`
`  IN ('Bizoi', 'Brouard')`
`ORDER BY JSON_DEPTH(doc) DESC;`
	`+----+------------+-----------------+`
	`| id | utilisateur | profondeur_maxi |`
	`+----+------------+-----------------+`
	`|  4 | Bizoi      |              5 |`
	`|  2 | Brouard    |              2 |`
	`|  1 | Brouard    |           NULL |`
	`+----+------------+-----------------+`

`SELECT id,`
`JSON_LENGTH(doc) AS taille,`
`JSON_LENGTH(doc,`
`'$.desc_vol.sieges') AS taille_sieges`
	`+----+--------+---------------+`
	`| id | taille | taille_sieges |`
	`+----+--------+---------------+`
	`|  1 |   NULL |          NULL |`
	`|  2 |      2 |          NULL |`

```
FROM   tab_vol_json3                        |  3 |      5 |             4 |
WHERE  id IN (1,2,3);                       +----+--------+---------------+
```

Pour obtenir des informations de types au niveau d'un champ en particulier, vous pourrez utiliser JSON_TYPE(*doc_json*), qui est capable de retourner, selon le cas, OBJECT, ARRAY, BOOLEAN, NULL ou le type SQL. La requête suivante retourne les types des différents champs des documents sélectionnés.

Tableau 7-85 Renseigner sur des types d'un contenu JSON

```
SELECT id,
   JSON_TYPE(doc) AS type_doc,
   JSON_TYPE(JSON_EXTRACT(doc, '$.sequence')) AS type_sequence,
   JSON_TYPE(JSON_EXTRACT(doc, '$.desc_vol')) AS type_desc_vol,
   JSON_TYPE(JSON_EXTRACT(doc, '$.desc_vol.sieges')) AS type_sieges,
   JSON_TYPE(JSON_EXTRACT(doc, '$.desc_vol.sieges[0].siege')) AS type_siege
FROM   tab_vol_json3
WHERE  id IN (1,4);
```

```
+----+----------+---------------+---------------+-------------+------------+
| id | type_doc | type_sequence | type_desc_vol | type_sieges | type_siege |
+----+----------+---------------+---------------+-------------+------------+
|  1 | NULL     | NULL          | NULL          | NULL        | NULL       |
|  4 | OBJECT   | INTEGER       | OBJECT        | ARRAY       | STRING     |
+----+----------+---------------+---------------+-------------+------------+
```

La fonction JSON_VALID(*expr*) permet de statuer sur la validité en termes de structure d'un document JSON. Il apparaît ainsi que deux documents sont valides et que la chaîne de caractères de la première ligne de la table n'est pas bien structurée au sens JSON.

Tableau 7-86 Renseigner sur la validité d'un contenu JSON

Instructions	Résultat

`SELECT id, utilisateur,`
` JSON_VALID(doc) AS doc_valide,`
` JSON_VALID(doc_chaine) AS chaine_`
`valide`
`FROM   tab_vol_json3`
`WHERE id < 4`
`ORDER BY id;`
	`+----+-------------+------------+---------------+`
	`| id | utilisateur | doc_valide | chaine_valide |`
	`+----+-------------+------------+---------------+`
	`|  1 | Brouard     |       NULL |             0 |`
	`|  2 | Brouard     |          1 |          NULL |`
	`|  3 | Salais      |          1 |          NULL |`
	`+----+-------------+------------+---------------+`

Mises à jour dans la table

C'est toujours l'instruction UPDATE qu'il faudra utiliser en combinant différentes fonctions permettant de composer du contenu JSON. L'instruction suivante permet de modifier la date d'un vol et d'ajouter un passager.

Tableau 7-87 Mises à jour d'un document

```
UPDATE tab_vol_json3
SET doc = (SELECT JSON_REPLACE(doc,
                              '$.desc_vol.sieges',
                              JSON_ARRAY_APPEND(
                                   JSON_EXTRACT(doc, '$.desc_vol.sieges'),
                                   '$',
                                    JSON_OBJECT('prenom','Antoine',
                                                'nom','Derouin','siege','5A')))),
doc = (SELECT JSON_REPLACE(doc, '$.date_vol', '2016-07-24'))
WHERE  id = 3;
```

```
mysql> SELECT id,
    ->        JSON_EXTRACT(doc,'$.num_vol','$.date_vol') AS elements_vol,
    ->        JSON_EXTRACT(doc,'$.desc_vol.sieges[*].siege') AS passagers
    -> FROM   tab_vol_json3
    -> WHERE  id = 3;
+----+--------------------------+----------------------------------------+
| id | elements_vol             | passagers                              |
+----+--------------------------+----------------------------------------+
|  3 | ["AF6143", "2016-07-24"] | ["15A", "15B", "15C", "13C", "5A", "5A"] |
+----+--------------------------+----------------------------------------+
```

Il est aussi possible de construire un document à partir d'une chaîne à l'aide de la fontion de conversion CAST.

Tableau 7-88 Conversion d'une chaîne en document

```
UPDATE tab_vol_json3
SET doc = CAST('{"date_vol":"2016-07-18","num_vol":"AF6140"}' AS JSON)
WHERE  id = 1;
```

```
mysql> SELECT id,doc FROM tab_vol_json3 WHERE id = 1;
+----+-------------------------------------------------+
| id | doc                                             |
+----+-------------------------------------------------+
|  1 | {"num_vol": "AF6140", "date_vol": "2016-07-18"} |
+----+-------------------------------------------------+
```

Opérateurs de comparaison

Des contenus JSON peuvent être comparés avec la plupart des opérateurs (=, <, <=, >, >=, <> et !=) mais pas avec les opérateurs BETWEEN, IN, GREATEST et LEAST. La comparaison s'opère en deux étapes. La première est basée sur les valeurs des types JSON, s'ils diffèrent le résultat dépendra de leur hiérarchie. Si les deux valeurs comparées sont de même type, alors le résultat dépendra de l'échelle du type en question. La hiérarchie des types est la suivante, du plus fort au plus faible : BLOB, BIT, OPAQUE, DATETIME, TIME, DATE, BOOLEAN, ARRAY, OBJECT, STRING, INTEGER/DOUBLE, NULL).

La requête suivante compare des fragments de contenus JSON, le premier est

un entier, le deuxième une chaîne, le troisième un objet et le troisième un tableau (le plus fort dans la précédence des types ici représentés).

Tableau 7-89 Comparaison de contenus

```
SELECT id,
     JSON_EXTRACT(doc, '$.sequence')           AS seq,
     JSON_EXTRACT(doc,'$.num_vol')             AS num_vol,
     JSON_TYPE(doc)                            AS type_doc,
     JSON_EXTRACT(doc,'$.num_vol','$.date_vol') AS elements_vol
FROM   tab_vol_json3
WHERE  id IN (3,4)
AND    JSON_EXTRACT(doc,'$.num_vol')             > JSON_EXTRACT(doc, '$.sequence')
AND    JSON_EXTRACT(doc,'$.num_vol','$.date_vol') > JSON_EXTRACT(doc,'$.num_vol')
AND    JSON_EXTRACT(doc,'$.num_vol','$.date_vol') > doc;
```

```
+----+------+----------+----------+-------------------------+
| id | seq  | num_vol  | type_doc | elements_vol            |
+----+------+----------+----------+-------------------------+
|  2 | NULL | "AF6141" | OBJECT   | ["AF6141", "2016-07-18"] |
|  3 | 345  | "AF6143" | OBJECT   | ["AF6143", "2016-07-19"] |
+----+------+----------+----------+-------------------------+
```

Les clauses `ORDER BY` et `GROUP BY` répondent également à ces règles.

Tableau 7-90 Tri sur du contenu

```
SELECT id,
       JSON_EXTRACT(doc,'$.desc_vol.porte')       AS num_porte,
       JSON_EXTRACT(doc,'$.num_vol','$.date_vol') AS elements_vol
FROM   tab_vol_json3
WHERE  id IN (2,3,4)
ORDER BY elements_vol;
```

```
+----+-----------+-------------------------+
| id | num_porte | elements_vol            |
+----+-----------+-------------------------+
|  2 | NULL      | ["AF6141", "2016-07-18"] |
|  3 | "20A"     | ["AF6143", "2016-07-19"] |
|  4 | "20A"     | ["AF6146", "2016-07-19"] |
+----+-----------+-------------------------+
```

Indexation

Depuis la version 5.7.8, et au niveau du moteur InnoDB, l'indexation secondaire sur des colonnes virtuelles est possible. Un index secondaire peut être créé à partir d'une ou de plusieurs colonnes virtuelles ou d'une combinaison de colonnes virtuelles ou non. Un index sur une colonne virtuelle peut être défini unique.

Le tableau suivant présente deux index uniques, le premier portant sur le champ *sequence*, l'autre sur le champ *_id*, qui est une chaîne de caractères et nécessite d'ôter les doubles quotes à chaque valeur.

Tableau 7-91 Création d'index sur du contenu JSON

Index sur un numérique	Index sur une chaîne

`ALTER TABLE tab_vol_json3`
`  ADD doc_sequence INT`
`  GENERATED ALWAYS AS`
`  (JSON_EXTRACT(doc, '$.sequence'));`
`CREATE UNIQUE INDEX index_doc_seq`
`  ON tab_vol_json3(doc_sequence);`
	`ALTER TABLE tab_vol_json3`
	`  ADD doc_id VARCHAR(8)`
	`  GENERATED ALWAYS AS`
	`  (JSON_UNQUOTE(`
	`  JSON_EXTRACT(doc, '$._id')));`
	`CREATE UNIQUE INDEX index_doc_id`
	`  ON tab_vol_json3(doc_id);`

Exercices

L'objectif de ces exercices est d'écrire des sous-programmes MySQL manipulant des curseurs et gérant des exceptions sur la base de données *Parc informatique*.

Exercice 7.1

Curseur

On désire connaître, pour chaque logiciel installé, le temps (nombre de jours entier décimal) passé entre l'achat et l'installation. Ce calcul devra renseigner la colonne `delai` de la table `Installer` pour l'instant nulle.

Utiliser une table `test.Trace(message VARCHAR(80)` (et l'afficher en fin de sous-programme) pour stocker :

les incohérences (date d'installation antérieure à la date d'achat, date d'installation ou date d'achat inconnue) ;

le nombre entier de jours séparant l'achat de l'installation (utiliser `DATEDIFF`) ;

une chaîne simulant un format *TIME* étendu qui représente le nombre de jours décimal séparant l'achat de l'installation (par exemple, si le nombre de jours décimal vaut « 14,5 », il faudra construire la chaîne : « `14 j 12:00:00` »).

Écrire la procédure `calculTemps` pour programmer ce processus. Un exemple de table `test.Trace` à produire en sortie :

```
+-------------------------------------------------------------+
| message                                                     |
+-------------------------------------------------------------+
| Logiciel Oracle 6 sur Poste 2 attente 2924 jour(s).          |
| En format TIME étendu 2924 j 00:00:00                        |
| Logiciel Oracle 8 sur Poste 2 attente 1463 jour(s).          |
```

```
| …                                                                |
| Logiciel I. I. S. installé sur Poste 7 11 jour(s) avant l'achat! |
| Date d'achat inconnue pour le logiciel SQL*Net sur Poste 2       |
| Logiciel Oracle 6 sur Poste 8 attente 3876 jour(s).              |
| En format TIME étendu 3876 j 10:59:17                            |
| …                                                                |
+------------------------------------------------------------------+
```

Exercice 7.2

Transaction

Écrire la procédure `installLogSeg` permettant d'effectuer une installation groupée sur tous les postes d'un même segment d'un nouveau logiciel. La transaction doit enregistrer dans un premier temps le nouveau logiciel, puis les différentes installations sur tous les postes du segment de même type que celui du logiciel acheté. L'installation se fera à la date du jour. Penser à mettre à jour la colonne `delai` comme programmé précédemment.

Ne pas encore tenir compte des éventuelles exceptions et tracer chaque insertion dans la table `test.Trace`. Utiliser les paramètres ci-dessous pour tester votre procédure. L'état de sortie doit être le suivant. Vérifier aussi la présence des deux nouveaux enregistrements dans la table `Installer`. Ne programmer le `COMMIT` qu'une fois la procédure bien testée.

```
CALL installLogSeg('130.120.80', 'log99', 'Blaster', '2005-09-05', '9.9', 'PCWS', 999.9)$
+-------------------------------------+
| message                             |
+-------------------------------------+
| Blaster stocké dans la table Logiciel |
| Installation sur Poste 4 dans Salle 2 |
| Installation sur Poste 5 dans Salle 2 |
+-------------------------------------+
```

Exercice 7.3

Exceptions

Modifier la procédure `installLogSeg` afin de prendre en compte quelques-unes des exceptions potentielles :

numéro de segment inconnu (erreur `NOT FOUND`) ;

numéro de logiciel déjà présent (`ERROR 1062` *Duplicate entry*) ;

type du logiciel inconnu (`ERROR 1452` *Cannot add or update a child row*) ;

date d'achat postérieure à celle du jour (se servir du même calcul que pour la

colonne `delai` de l'exercice précédent) ;

aucune installation réalisée, car pas de poste de travail de ce type (erreur utilisateur `pas_install_possible`).

Vérifier chacun de ces cas avec le jeu de tests suivant :

```
--test segment
CALL installLogSeg(' tot', 'log99','Blaster', '2005-09-05', '9.9', 'PCWS', 999.9)$
--test logiciel déjà présent
CALL installLogSeg('130.120.80', ' log','Blaster', '2005-09-05', '9.9', 'PCWS', 999.9)$
--test type du logiciel
CALL installLogSeg('130.120.80', 'log98','Mozilla', '2005-11-04', '1', ' tot', 100.0)$
--date d'achat plus grande que celle du jour ?
CALL installLogSeg('130.120.80', 'log98','Mozilla', '2010-11-04', '1', 'PCWS', 100.0)$
--aucune install
CALL installLogSeg( '130.120.81', 'log55','Eudora', '2005-12-06', '5', 'PCWS', 540)$
--bonne installation
CALL installLogSeg('130.120.80', 'log77','Blog Up', '2005-12-05', '1.3', 'PCWS', 90)$
```

Exercice 7.4

Déclencheurs

Mises à jour de colonnes

Écrire les déclencheurs `Trig_AD_Installer` et `Trig_AI_Installer` sur la table `Installer` permettant de faire la mise à jour automatique des colonnes `nbLog` de la table `Poste`, et `nbInstall` de la table `Logiciel`. Prévoir les cas de désinstallation d'un logiciel (`AFTER DELETE`) sur un poste, et d'installation (`AFTER INSERT`) d'un logiciel sur un autre.

Écrire les déclencheurs `Trig_AI_Poste` et `Trig_AD_Poste` sur la table `Poste` permettant d'actualiser la colonne `nbPoste` de la table `Salle` à chaque ajout ou suppression d'un nouveau poste.

Écrire le déclencheur `Trig_AU_Salle` sur la table `Salle` qui met à jour automatiquement la colonne `nbPoste` de la table `Segment` après la modification de la colonne `nbPoste`.

Ces deux derniers déclencheurs vont s'enchaîner : l'ajout ou la suppression d'un poste entraînera l'actualisation de la colonne `nbPoste` de la table `Salle`, qui conduira à la mise à jour de la colonne `nbPoste` de la table `Segment`. Ajouter un poste pour vérifier le rafraîchissement des deux tables (`Salle` et `Segment`). Supprimer ce poste puis vérifier à nouveau la cohérence des deux tables.

Programmation de contraintes

Écrire le déclencheur `Trig_BI_Installer` sur la table `Installer` permettant de contrôler, avant chaque nouvelle installation, que le type du logiciel correspond au type du poste, et que la date d'installation est soit nulle soit postérieure à la date d'achat.

Partie III

Langages et outils

Chapitre 8

Utilisation avec Java

MySQL offre, sur son site, différents pilotes pour rendre compatibles des applications avec une base de données sur différents systèmes.

- *Connector/ODBC* (*Open DataBase Connectivity*) pour Windows, Linux, Mac OS X, et Unix ;
- *Connector/J* pour toute plate-forme Java en utilisant JDBC (*Java DataBase Connectivity*) ;
- *Connector/Net* pour toute plate-forme .Net ;
- Connector/MXJ : composant qui encapsule le moteur MySQL dans une application J2EE.

Ce chapitre explique l'utilisation de l'API JDBC 3.0 pour manipuler une base MySQL via un programme Java.

JDBC avec Connector/J

L'interface JDBC initialement programmée par Sun, appelée aussi « passerelle » ou « API », est composée d'un ensemble de classes permettant le dialogue entre une application Java et une source de données compatible SQL (tables relationnelles en général, mais aussi données issues d'un fichier texte ou d'un classeur Excel par exemple). L'API JDBC 3.0 que MySQL fournit gratuitement est appelée *Connector/J*.

L'interface JDBC est conforme au niveau d'entrée de la norme SQL2 (*entry level*) et prend en charge la programmation *multithread*. La communication est réalisée en mode client-serveur déconnecté et s'effectue en plusieurs étapes :

- connexion à la base de données ;
- émissions d'instructions SQL et exploitation des résultats provenant de la base de données ;

- déconnexion de la base.

Le spectre de JDBC est large, car l'applicatif Java peut être une classe ou une *applet* côté client, une *servlet*, un EJB (*Enterprise Java Beans*) ou une procédure cataloguée côté serveur.

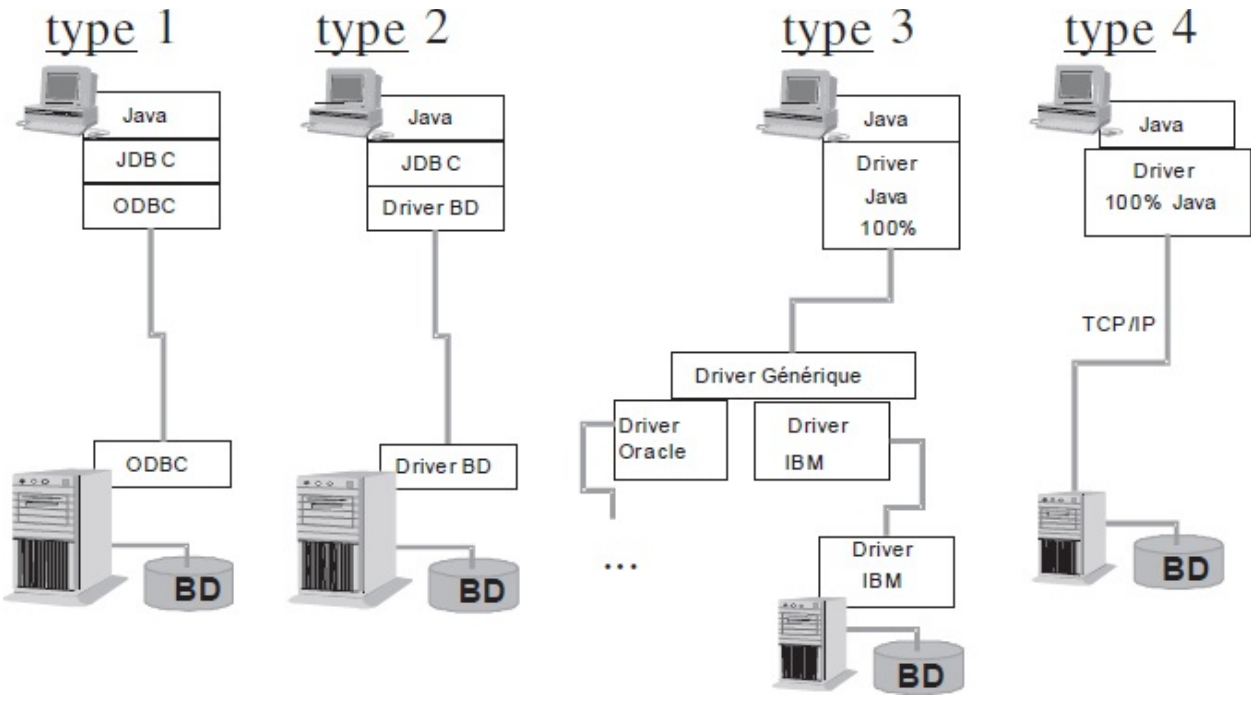
Classification des pilotes [drivers]

Un pilote (*driver*) JDBC est une couche logicielle chargée d'assurer la liaison entre l'application Java (cliente) et le SGBD (serveur). La classification des pilotes JDBC distingue quatre types :

- Les pilotes de type 1 (*JDBC-ODBC Bridge*) utilisent la couche logicielle de Microsoft appelée ODBC. Le client est dit « épais », puisque le pilote JDBC convertit les appels Java en appels ODBC avant de les exécuter. Cette approche convient bien pour des sources de données Windows ou si l'interface cliente est écrite dans un langage natif de Microsoft.
- Les pilotes de type 2 (*Native-API Partly-Java Driver*) utilisent un pilote fourni par le constructeur de la base de données (natif). Le pilote n'étant pas développé en Java, le client est aussi dit « épais » pour cette approche. En effet, les commandes JDBC sont toutes converties en appels natifs du SGBD considéré. Cette approche convient pour les applications qui manipulent des sources de données uniques (tout Oracle ou IBM, etc.).
- Les pilotes de type 3 (*Net Protocol All-Java Driver*) utilisent un pilote générique natif écrit en Java. Le client est plus « léger » car les appels JDBC sont transformés par un protocole indépendant du SGBD. Cette approche convient pour des sources de données hétérogènes.
- Les pilotes de type 4 (*Native Protocol All-Java Driver*) sont écrits en Java. Le client est léger car il ne nécessite aucune couche logicielle supplémentaire. Les appels JDBC sont traduits en *sockets* exploités par le SGBD. Cette approche est la plus simple, mais pas forcément la plus puissante ; elle convient pour tous les types d'architectures.

La figure suivante schématise le principe mis en œuvre au travers des quatre types de pilotes JDBC :

Figure 8-1 *Types de pilotes JDBC*



Le choix du pilote n'a pas d'influence majeure sur la programmation. Seules les phases de chargement du pilote et de connexion aux bases sont spécifiques, les autres instructions sont indépendantes du pilote. En d'autres termes, si vous avez une application déjà écrite et que vous décidez de changer le type du pilote – soit que la source de données migre de MySQL à Access, à Oracle ou à SQL Server par exemple, soit que vous optiez pour un autre pilote en conservant votre source de données –, seules quelques instructions devront être réécrites.

Avec MySQL, vous pouvez travailler avec l'API de Sun, mais vous n'avez pas trop le choix pour le type du pilote (*Connector/J* est un pilote JDBC de type 4).

Le paquetage `java.sql`

La version 3.0 de JDBC est composée de classes et d'interfaces situées dans le paquetage `java.sql` du JDK. MySQL propose également une API propriétaire (qui redéfinit et étend celle de Sun). Le tableau suivant résume la composition de ce paquetage.

Tableau 8-1 L'API JDBC 3.0 standard

Classe/interface	Description

`java.sql.Driver` `java.sql.Connection`	Pilotes JDBC pour les connexions aux sources de données SQL.

`java.sql.Statement`
`java.sql.PreparedStatement`
`java.sql.CallableStatement`
	Construction d'ordres SQL.

`java.sql.ResultSet`	Gestion des résultats des requêtes SQL.
`java.sql.DriverManager`	Gestion des pilotes de connexion.
`java.sql.SQLException`	Gestion des erreurs SQL.
`java.sql.DatabaseMetaData` `java.sql.ResultSetetaData`	Gestion des méta-informations (description de la base de données, des tables…).
`java.sql.SavePoint`	Gestion des transactions et des sous-transactions.

Structure d'un programme

La structure d'un programme Java utilisant JDBC comprend successivement les phases :

- d'importation de paquetages ;
- de chargement d'un pilote ;
- de création d'une ou de plusieurs connexions ;
- de création d'un ou de plusieurs états ;
- d'émission d'instructions SQL sur ces états ;
- de fermeture des objets créés.

Le code suivant (`JDBCTest.java`) décrit la syntaxe du plus simple programme JDBC. Nous inscrivons toutes les phases dans un même bloc (le `main`), mais elles peuvent se trouver dans différents blocs ou méthodes de diverses classes.

Tableau 8-2 Programme de test de connexion JDBC

Code Java	Commentaires
`import java.sql.*;`	Importation du paquetage.

`public class JDBCTest`
`{public static void main(String[] args)`
`throws SQLException, Exception`
`{`
	Classe ayant une méthode `main`.

```
    try
    { System.out.println
          ("Initialisation de la connexion");
    Class.forName
          ("com.mysql.jdbc.Driver").newInstance();
    } catch (ClassNotFoundException ex)
       { System.out.println
         ("Problème au chargement"+ex.toString()); }
    try
    { Connection cx = DriverManager.getConnection
            ("jdbc:mysql://localhost/bdsoutou?
              user=soutou&password=iut") ;

      Statement etat = cx.createStatement ();
      ResultSet rset = etat.executeQuery
          ("SELECT SYSDATE()");
      while (rset.next ())
            System.out.println("Nous sommes le : "+

                               rset.getString (1));
      System.out.println("JDBC correctement
                                configuré");
    }
    catch(SQLException ex)
         { System.err.println("Erreur : "+ex); }
  }
}
```

Chargement du pilote JDBC MySQL.

Création d'une connexion.

Création d'un état de connexion.
Extraction de la date courante.

Affichage du résultat.

Gestion des erreurs.

Le dernier bloc permet de récupérer les erreurs renvoyées par le SGBD. Nous détaillerons en fin de section le traitement des exceptions.

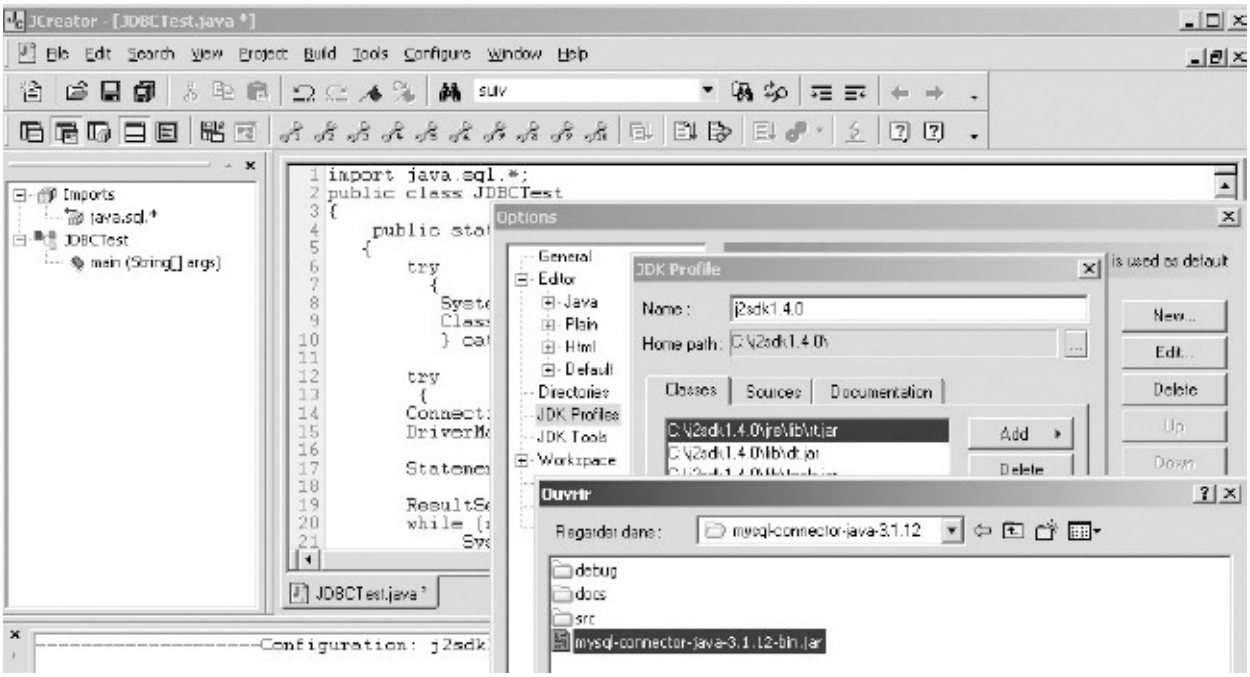
Test de votre configuration

L'environnement JDBC sous MySQL nécessite la configuration d'un certain nombre de variables :

- La variable `PATH` doit contenir le chemin de la machine virtuelle Java pour compiler et exécuter des classes. Le JDK est en général installé dans `C:\j2sdk1.XXX`, les fichiers `javac` et `java` se trouvent dans le sous-répertoire `bin`.
- La variable `CLASSPATH` doit inclure le paquetage JDBC (fichier `.jar`) pour MySQL (téléchargeable sur le site de MySQL). Pour ma part, j'ai dézippé le fichier `mysql-connector-java-3.1.12.zip` dans le répertoire `C:\temp`.

Vous pouvez tester votre environnement en utilisant le fichier `JDBCTest.java`. Si vous utilisez l'outil *JCreator*, configurez la variable `CLASSPATH` de la manière suivante : `Configure`/`Options`/`JDK Profiles`, clic sur la version du JDK, puis `Edit`, onglet `Classes`, faire `Add Archive` et choisir le fichier `jar` (pour mon cas `mysql-connector-java-3.1.12-bin.jar`).

***Figure 8-2** Interface JCreator*



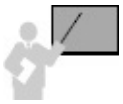
Cet exemple décrit le code nécessaire à la connexion à votre base (il faudra modifier le nom de la base, le nom et le mot de passe de l'utilisateur) et doit renvoyer les messages suivants :

```
Initialisation de la connexion
Nous sommes le : date et heure courante
JDBC correctement configuré
```

Connexion à une base

La connexion à une base de données est rendue possible par l'utilisation de la classe `Driver-`

`Manager` et de l'interface `Connection`.



Deux étapes sont nécessaires pour qu'un programme Java puisse se connecter à une base de données :

- Le chargement du pilote par appel de la méthode `java.lang.Class.forName`.
- L'établissement de la connexion en créant un objet (ici `cx`) de l'interface

`Connection` par l'instruction suivante : `cx = DriverManager.getConnection(`*`chaîneConnexion`*`);`

Pour MySQL, nous verrons que le paramètre *`chaîneConnexion`* représente une variable dont une syntaxe simplifiée est de type :

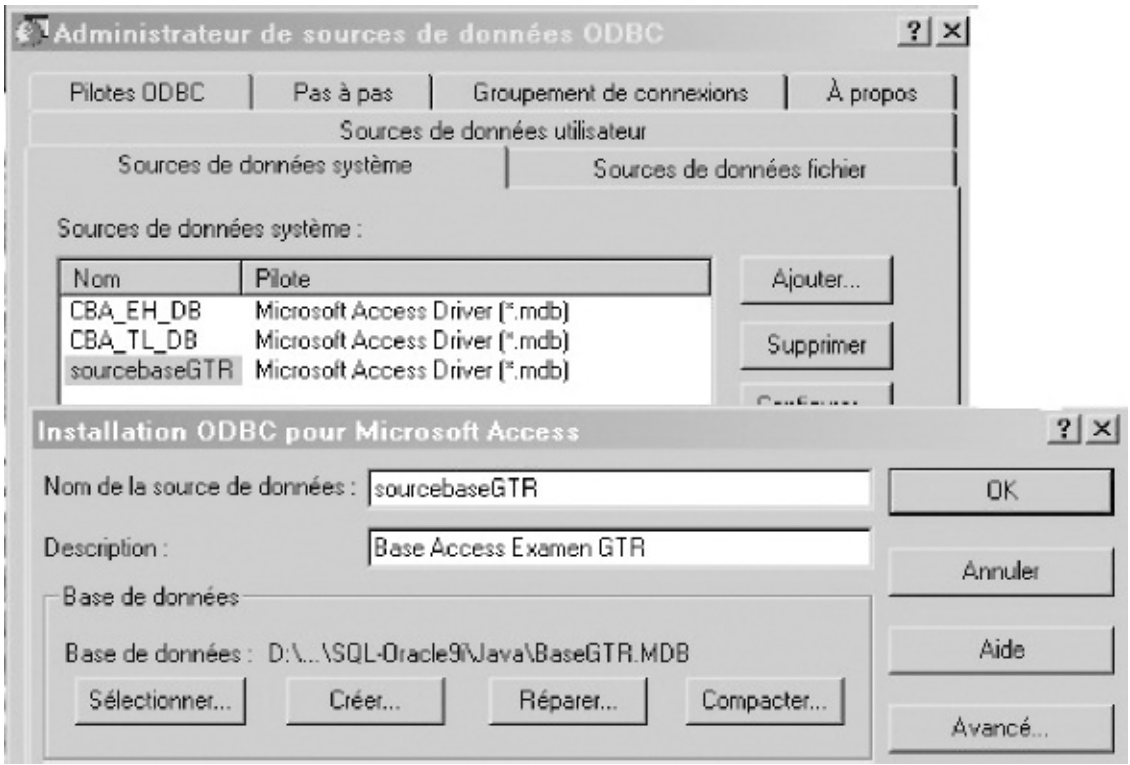
```
jdbc:mysql://[host][:port]/[database][?user][=nomUtil][&password][=motPasse]
```

Cette chaîne permettra de désigner la base et d'identifier l'utilisateur.

Base Access

Étudions, pour information, l'établissement de la connexion d'un pilote de type 1 pour se mettre en rapport avec une base *Access* via une source de données ODBC. La figure suivante illustre les parties du panneau de configuration Windows qui permettent de désigner une base *Access*. Dans notre exemple, la source (`BaseGTR.MDB`) est située dans un répertoire sous l'unité de disque `D:\` et désignée par le DSN (*Data Source Name*) `sourcebaseGTR` :

***Figure 8-3** Source de données ODBC*



Le code suivant (`TestJDBCODBC.java`) charge un pilote de type 1, puis se connecte à la source ODBC précitée (inutile de préciser le nom et le mot de passe de l'utilisateur du fait d'une base *Access*). Le DSN est noté en gras dans le script.

Tableau 8-3 Programme JDBC

Code Java	Commentaires
`import java.sql.*;`	Importation.
`class TestJDBCODBC` `{  public static void main (String args [])`	Classe ayant une méthode `main`.
`throws SQLException`	Chargement d'un pilote JDBC/ODBC.

`{try {Class.forName("sun.jdbc.odbc.JdbcOdbcDriver"); }`
`catch (ClassNotFoundException ex)`
`{ System.out.println ("Problème au chargement"); }`	

`try {Connection conn = DriverManager.getConnection` `("jdbc:odbc:`**`sourcebaseGTR`**`","","");`	Connexion à la base Access.
`...}`	
`catch(SQLException ex){ … } } }`	Gestion des erreurs.

Base MySQL

Seules les phases de chargement de pilote et de création de la connexion changent. Afin de transférer un pilote MySQL, il faut utiliser l'interface `DriverManager` implémentée par l'appel de la méthode `Class.forName()`. Sous *Connector/J*, le nom de la classe à charger est `com.mysql.jdbc.Driver`. La connexion s'effectue par la méthode `getConnection`.

Tableau 8-4 Chargement du pilote MySQL

Code Java	Commentaires

`try`
`{ `**`Class.forName`**`("com.mysql.jdbc.Driver").newInstance();`
`} catch (ClassNotFoundException ex)`
`{ System.out.println`
`("Problème au chargement"+ex.toString()); }`
	Chargement du pilote MySQL.

`try`
`{ `**`Connection`**` cx = `**`DriverManager`**`.getConnection`
`("jdbc:mysql://localhost/bdsoutou?`
	Déclaration d'une

connexion.

```
            user=soutou&password=iut");  ... }
catch(SQLException ex)
  { System.err.println("Erreur : "+ex); }
```

Gestion des erreurs.

Interface `Connection`

Le tableau ci-après présente les principales méthodes disponibles de l'interface `Connection`. Nous détaillerons l'invocation de certaines de ces méthodes à l'aide des exemples des sections suivantes.

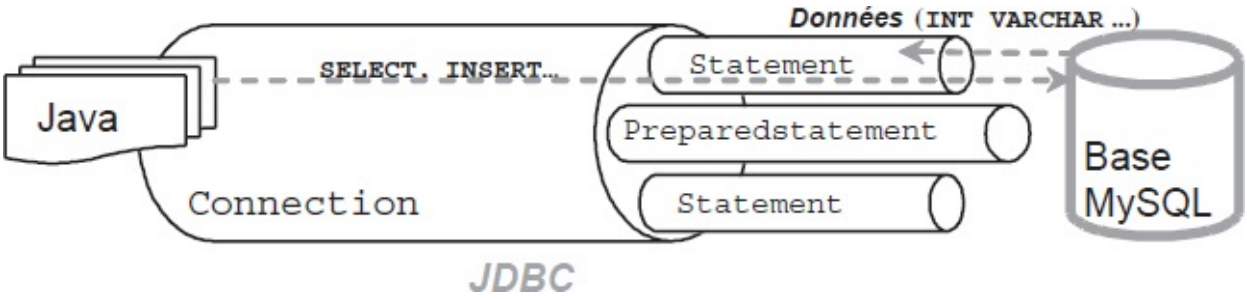
Tableau 8-5 Méthodes de l'interface Connection

Méthode	Description
`createStatement()`	Création d'un objet destiné à recevoir un ordre SQL statique, non paramétré.
`prepareStatement(String)`	Précompile un ordre SQL acceptant des paramètres et pouvant être exécuté plusieurs fois.
`prepareCall(String)`	Appel d'une procédure cataloguée (certains pilotes attendent `execute` ou ne reconnaissent pas `prepareCall`).
`void setAutoCommit(boolean)`	Positionne ou non le *commit* automatique.
`void commit()`	Valide la transaction.
`void rollback()`	Invalide la transaction.
`void close()`	Ferme la connexion.

États d'une connexion

Une fois la connexion établie, il est nécessaire de définir des états qui permettront l'encapsulation d'instructions SQL dans du code Java. Un état permet de faire passer plusieurs instructions SQL sur le réseau. On peut affecter à un état une ou plusieurs instruction SQL. Si on désire exécuter plusieurs fois la même instruction, il est intéressant de réserver l'utilisation d'un état à cet effet.

Figure 8-4 *Connexion et états*



Interfaces disponibles

Différentes interfaces sont prévues à cet effet :

- `Statement` pour les ordres SQL statiques. Ces états sont construits par la méthode `createStatement` appliquée à la connexion.
- `PreparedStatement` pour les ordres SQL paramétrés. Ces états sont construits par la méthode `prepareStatement` appliquée à la connexion.
- `CallableStatement` pour les procédures ou fonctions cataloguées. Ces états sont construits par la méthode `prepareCall` appliquée à la connexion.

S'il ne doit plus être utilisé dans la suite du code Java, chaque objet de type `Statement`, `PreparedStatement` ou `CallableStatement` devra être fermé à l'aide de la méthode `close`.

Méthodes génériques pour les paramètres

Une fois qu'un état est créé, il est possible de lui passer des paramètres par des méthodes génériques (étudiées plus en détail par la suite) :

- `setxxx` où *xxx* désigne le type de la variable (exemple : `setString` ou `setInt`) du sens Java vers MySQL (*setter methods*). Il s'agit ici de paramétrer un ordre SQL (instruction ou appel d'un sous-programme) ;
- `getxxx` (exemple : `getString` ou `getInt`) du sens MySQL vers Java. Il s'agit ici d'extraire des données de la base dans des variables hôtes Java via un curseur Java (*getter methods*) ;
- `updatexxx` (exemple : `updateString` ou `upadateInt`) du sens Java vers MySQL. Il s'agit ici de mettre à jour des données de la base via un curseur Java (*updater methods*). Ces méthodes sont disponibles seulement depuis la version 2 de JDBC (SDK 1.2).

États simples [interface `Statement`]

Nous décrivons ici l'utilisation d'un état simple (interface `Statement`). Nous étudierons par la suite les instructions paramétrées (interface `PreparedStatement`) et appels de sous-programmes (interface `CallableStatement`). Le tableau suivant décrit les principales méthodes de l'interface `Statement`.

Tableau 8-6 Méthodes de l'interface Statement

Méthode	Description
`ResultSet executeQuery(String)`	Exécute une requête et retourne un ensemble de lignes (objet `ResultSet`).
`int executeUpdate(String)`	Exécute une instruction SQL et retourne le nombre de lignes traitées (`INSERT`, `UPDATE` ou `DELETE`) ou 0 pour les instructions ne renvoyant aucun résultat (LDD).
`boolean execute(String)`	Exécute une instruction SQL et renvoie `true` si c'est une instruction `SELECT`, `false` sinon (instructions LMD ou plusieurs résultats `ResultSet`).
`Connection getConnection()`	Retourne l'objet de la connexion.
`void setMaxRows(int)`	Positionne la limite du nombre d'enregistrements à extraire par toute requête issue de cet état.
`int getUpdateCount()`	Nombre de lignes traitées par l'instruction SQL (-1 si c'est une requête ou si l'instruction n'affecte aucune ligne).
`void close()`	Ferme l'état.

Le code suivant (`Etats.java`) présente quelques exemples d'utilisation de ces méthodes sur un état (objet `etatSimple`). Nous supposons qu'un pilote JDBC est chargé et que la connexion `cx` a été créée. Nous verrons en fin de chapitre comment traiter proprement les exceptions.

Tableau 8-7 États simples

Code Java	Commentaires
`Statement etatSimple = cx.createStatement();`	Création de l'état.

`etatSimple.execute ("CREATE TABLE IF NOT EXISTS Compagnie(comp VARCHAR(4), nomComp VARCHAR(30), CONSTRAINT pk_Compagnie PRIMARY KEY(comp))");`	Ordre LDD.
`int j = etatSimple.executeUpdate ("CREATE TABLE IF NOT EXISTS Avion (immat VARCHAR(6),typeAvion VARCHAR(15), cap SMALLINT, compa VARCHAR(4), CONSTRAINT pk_Avion PRIMARY KEY(immat), CONSTRAINT fk_Avion_comp_Compagnie FOREIGN KEY(compa) REFERENCES Compagnie(comp))");`	Ordre LDD (autre écriture), *j* contient 0 (aucune ligne n'est concernée).
`int k = etatSimple.executeUpdate ("INSERT INTO Compagnie VALUES ('AF','Air France')");`	Ordre LMD, *k* contient 1 (une ligne est concernée).
`etatSimple.execute ("INSERT INTO Avion VALUES ('F-WTSS','Concorde',90,'AF')");` `etatSimple.execute("INSERT INTO Avion VALUES ('F-FGFB','A320',148,'AF')");`	Ordres LMD (autres écritures).
`etatSimple.setMaxRows (10);`	Pas plus de 10 lignes retournées par les prochaines extractions.
`ResultSet curseurJava = etatSimple.executeQuery ("SELECT * FROM Avion");`	Chargement d'un curseur Java.
`etatSimple.execute("DELETE FROM Avion");` `int l = etatSimple.getUpdateCount ();`	Ordre LMD, *l* contient 2 (avions supprimés).

Méthodes à utiliser

Le tableau suivant indique la méthode préférentielle à utiliser sur l'état courant (objet `Statement`) en fonction de l'instruction SQL à émettre :

Tableau 8-8 Méthodes Java pour les ordres SQL

Instruction SQL	Méthode	Type de retour
CREATE ALTER DROP	executeUpdate	int
INSERT UPDATE DELETE	executeUpdate	int
SELECT	executeQuery	ResultSet

Correspondances de types

Les échanges de données entre variables Java et colonnes des tables Oracle

impliquent de prévoir des conversions de types. D'une manière générale, tout type de donnée MySQL peut être convertit en un type `java.lang.String`. Les types numériques trouvent aussi une correspondance dans les types numériques Java (attention toutefois aux arrondis, dépassement de capacité ou perte de précision).

Les tableaux suivants présentent les principales correspondances existantes :

Tableau 8-9 Conversions possibles entre types

Les types MySQL	Peuvent créer les classes Java
`CHAR`, `VARCHAR`, `BLOB`, `TEXT`, `ENUM` et `SET`	`java.lang.String`, `java.io.InputStream`, `java.io.Reader`, `java.sql.Blob` et `java.sql.Clob`
`FLOAT`, `REAL`, `DOUBLE PRECISION`, `NUMERIC`, `DECIMAL`, `TINYINT`, `SMALLINT`, `MEDIUMINT`, `INTEGER` et `BIGINT`	`java.lang.String`, `java.lang.Short`, `java.lang.Integer`, `java.lang.Long`, `java.lang.Double` et `java.math.BigDecimal`
`DATE`, `TIME`, `DATETIME` et `TIMESTAMP`	`java.lang.String`, `java.sql.Date` et `java.sql.Timestamp`

La méthode `getObject()`de l'interface `ResultSet` (que nous allons étudier plus loin) réalise implicitement les conversions suivantes :

Tableau 8-10 Correspondances entre types

Types MySQL	Types Java
`BOOL` et `BOOLEAN`	alias de `TINYINT(1)`
`BLOB`, `BINARY(`*n*`)`, `BIT(>1)`, `LONGBLOB`, `MEDIUMBLOB`, `TINYBLOB` et `VARBINARY`(n)	`byte[]`
`BIT(1)` et `TINYINT`	`java.lang.Boolean` si la configuration `tinyInt1isBit` est mise à *true* (par défaut) et la taille de la variable = 1 (`java.lang.Integer` sinon).
`DOUBLE[(`*n*`,`*p*`)]`	`java.lang.Double`
`FLOAT[(`*n*`,`*p*`)]`	`java.lang.Float`
`SMALLINT[(`*n*`)] [UNSIGNED]`	`java.lang.Integer` (sans contrôle du signe).
`INT` et `INTEGER[(`*n*`)] [UNSIGNED]`	`java.lang.Integer` (si `UNSIGNED java.lang.Long`).
`MEDIUMINT[(`*n*`)] [UNSIGNED]`	`java.lang.Integer` (si `UNSIGNED java.lang.Long`).
`BIGINT[(`*n*`)] [UNSIGNED]`	`java.lang.Long`, (si `UNSIGNED java.math.BigInteger`).

TINYTEXT, TEXT, MEDIUMTEXT, LONGTEXT, ENUM(...) et SET(...)	java.lang.String
CHAR(M), VARCHAR(M) [BINARY]	java.lang.String (si BINARY, byte[]).
DECIMAL[(*n*[,*p*])]	java.math.BigDecimal
DATE	java.sql.Date
YEAR[(2|4)]	java.sql.Date (date initialisée au 1er Janvier 0h)
TIME	java.sql.Time
DATETIME, TIMESTAMP[(*n*)]	java.sql.Timestamp

Il est possible de connaître le type de la variable (ou de l'objet Java) que vous devez utiliser dans votre programme JDBC pour travailler avec une colonne *col* d'une table *tab*. Pour ce faire, utiliser la méthode `Class getClass()` appliqué à l'objet résultant de l'extraction de la colonne, dans le `ResultSet`, par `getObject("col")`. Une fois cette classe instanciée, il reste à utiliser `String getName()` pour trouver son nom.

Le code suivant (`CorresTypes.java`) présente la manière d'extraire le type Java nécessaire pour travailler avec la colonne `cap` de la table `Avion` (ici `SMALLINT`).

Tableau 8-11 Déduction du type Java à utiliser

Code Java	Commentaires
`ResultSet curseurJava = etatSimple.executeQuery ("SELECT cap FROM Avion LIMIT 1");`	Extraction dans un curseur de la capacité du premier avion.
`while (curseurJava.next ())`	Ouverture et lecture du curseur.
`{ Object obj = curseurJava.getObject("cap");`	
`System.out.println("Valeur : " +curseurJava.getObject("cap"));`	Extraction de l'objet Java équivalent.
`Class clJava = obj.getClass();`	
`System.out.println ("Classe Java equivalente : " +clJava.getName());}`	Déduction de sa classe.

La trace de ce programme est la suivante :

```
Valeur : 148
Classe Java equivalente :  java.lang.Integer
```

Manipulations avec la base

Détaillons à présent les différents scénarios que l'on peut rencontrer lors d'une manipulation de la base de données par un programme Java. Les tableaux suivants répertorient les conséquences les plus fréquentes. Les autres cas (relatifs aux contraintes référentielles et aux problèmes de syntaxe) seront étudiés dans la section « Traitement des exceptions ».

Suppression de données

Tableau 8-12 Enregistrements présents dans la table

Code Java	Résultat
`etat.executeUpdate("DELETE FROM Avion");`	Fait la suppression et passe en séquence.
`j = etat.executeUpdate("DELETE FROM Avion");`	Fait la suppression, affecte à `j` le nombre d'enregistrements supprimés et passe en séquence.

Tableau 8-13 Aucun enregistrement dans la table

Code Java	Résultat
`etat.executeUpdate("DELETE FROM Avion");`	Aucune action sur la base et passe en séquence.
`j = etat.executeUpdate("DELETE FROM Avion");`	Aucune action sur la base, affecte à `j` la valeur 0 et passe en séquence.

Ajout d'enregistrements

Tableau 8-14 Différentes écritures d'un `INSERT`

Code Java	Résultat
`etat.executeUpdate("INSERT INTO Compagnie VALUES ('TAF','Toulouse Air Free')");`	Fait l'insertion et passe en séquence.
	Fait l'insertion, affecte à `j` le

`int j= etat.executeUpdate("INSERT INTO Compagnie VALUES ('TAF','Toulouse Air Free')");`	nombre 1 et passe en séquence.

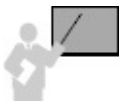
Modification d'enregistrements

Tableau 8-15 Différentes écritures d'un `UPDATE`

Code Java	Résultat
`etat.executeUpdate("UPDATE Compagnie SET nomComp = 'Air France Compagny' WHERE comp = 'AF'");`	Fait la modification et passe en séquence. Si aucun enregistrement n'est concerné, aucune exception n'est levée.
`int j= etat.executeUpdate("UPDATE Avion SET capacite=capacite*1.2");`	Fait la (les) modification(s), affecte à `j` le nombre d'enregistrements modifiés et passe en séquence (0 si aucun enregistrement n'est modifié).

Extraction de données

Étudions ici la gestion des résultats d'une instruction `SELECT`.



Le résultat d'une requête est placé dans un objet de l'interface `ResultSet` qui s'apparente à un curseur Java.

Le tableau suivant présente les principales méthodes disponibles de l'interface `ResultSet`. Les méthodes relatives aux curseurs navigables seront étudiées par la suite. Le parcours de ce curseur s'opère par la méthode `next`. Initialement (après création et chargement du curseur), on est positionné avant la première ligne. Bien qu'un objet de l'interface `ResultSet` soit automatiquement fermé quand son état est fermé ou recréé, il est préférable de le fermer explicitement par la méthode `close` s'il ne doit pas être réutilisé.

Tableau 8-16 Méthodes principales de l'interface ResulSet

Méthode	Description
`boolean next()`	Charge l'enregistrement suivant en retournant `true`, renvoie `false` lorsqu'il n'y a plus d'enregistrement suivant.
`void close()`	Ferme le curseur.
`get`*`XXX`*`(int)`	Récupère, au niveau de l'enregistrement, la valeur de la colonne numérotée de type *xxx*. Exemple : `getInt`(1), `getString`(1), `getDate`(1), etc., pour récupérer la valeur de la première colonne.
`update`*`XXX`*`(…)`	Modifie, au niveau de l'enregistrement, la valeur de la colonne numérotée de type *xxx*. Exemple : `updateInt`(1,i), `updateString`(1,`nom`), etc.
`ResultSetMetaData getMetaData()`	Retourne un objet `ResultSetMetaData` correspondant au curseur.
`Object getObject(String)`	Retourne la valeur de la colonne désignée par le paramètre dans une variable Java de type adéquat.

Distinguons l'instruction `SELECT` qui génère un curseur statique (objet `Resultset` utilisé sans option particulière) de celle qui produit un curseur navigable ou modifiable (objet `Resultset` employé avec des options disponibles depuis la version 2 de JDBC).

Curseurs statiques

Le code suivant (`SELECTstatique.java`) extrait les avions de la compagnie 'Air France' par l'intermédiaire du curseur `curseurJava`. Notez l'utilisation des différentes méthodes `get` pour récupérer des valeurs issues de colonnes.

Tableau 8-17 Extraction de données dans un curseur statique

Code Java	Commentaires

`try {…`
`Statement etatSimple = cx.createStatement();`
`ResultSet curseurJava`
	Création de l'état.

`etatSimple.executeQuery ("SELECT immat, cap FROM Avion WHERE comp = (SELECT comp FROM Compagnie WHERE nomComp='Air France')");`
`float moyenneCapacité =0;`
`int nbAvions = 0;`
	Création et chargement du curseur.

`while (curseurJava.next())` `{System.out.print("Immat : "+curseurJava.getString(1));`	Parcours du curseur.

`System.out.println(Capacité : "+curseurJava.getInt(2));`
`moyenneCapacité += curseurJava.getInt(2);`
`nbAvions ++; }`
	Extraction de colonnes.

`moyenneCapacité /= nbAvions;` `System.out.println("Capacité moy : "+moyenneCapacité);`	
`curseurJava.close();`	Fermeture du curseur.
`} catch(SQLException ex) { … }`	Gestion des erreurs.

Curseurs navigables

Un curseur `ResultSet` déclaré sans option n'est ni navigable ni modifiable. Seul un déplacement du début vers la fin (par la méthode `next`) est admis. Il est possible de rendre un curseur navigable en permettant de le parcourir en avant ou en arrière, et en autorisant l'accès direct à un enregistrement d'une manière absolue (en partant du début ou de la fin du curseur) ou relative (en partant de la position courante du curseur). On peut aussi rendre un curseur modifiable (la base pourra être changée par l'intermédiaire du curseur).

Dès l'instant où on déclare un curseur navigable, il faut aussi statuer sur le fait qu'il soit modifiable ou pas (section suivante). La nature du curseur est explicitée à l'aide d'options de la méthode `createStatement` :

```
Statement createStatement(int typeCurseur, int modifCurseur)
```

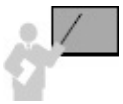
Constantes

Les valeurs permises du premier paramètre (*`typeCurseur`*), et qui concernent le sens de parcours, sont présentées dans le tableau suivant :

Tableau 8-18 Constantes de navigation d'un curseur

Constante	Explication
`ResultSet.TYPE_FORWARD_ONLY`	Le parcours du curseur s'opère invariablement du début à la fin (non navigable).

`ResultSet.TYPE_SCROLL_INSENSITIVE`	Le curseur est navigable mais pas sensible aux modifications.
`ResultSet.TYPE_SCROLL_SENSITIVE`	Le curseur est navigable et sensible aux modifications.



Un curseur est sensible dès que des mises à jour de la table sont automatiquement répercutées au niveau du curseur durant la transaction. Lorsqu'il est déclaré insensible, les modifications de la table ne sont pas renvoyées dans le curseur.

Méthodes

Les principales méthodes que l'on peut appliquer à un curseur navigable sont les suivantes. Les deux premières sont aussi des méthodes de l'interface `Statement` qui affectent et précisent le sens de parcours pour tous les curseurs de l'état donné.

Tableau 8-19 Méthodes de navigation dans un curseur

Méthode	Fonction
`void setFetchDirection(int)`	Affecte la direction du parcours : `ResultSet.FETCH_FORWARD (1000)`, `ResultSet.FETCH_REVERSE (1001)` ou `ResultSet.FETCH_UNKNOWN (1002)`.
`int getFetchDirection()`	Extrait la direction courante (une des trois valeurs ci-dessus).
`boolean isBeforeFirst()`	Indique si le curseur est positionné avant le premier enregistrement (`false` si aucun enregistrement n'existe).
`void beforeFirst()`	Positionne le curseur avant le premier enregistrement (aucun effet si le curseur est vide).
`boolean isFirst()`	Indique si le curseur est positionné sur le premier enregistrement (`false` si aucun enregistrement n'existe).

`boolean isLast()`	Indique si le curseur est positionné sur le dernier enregistrement (`false` si aucun enregistrement n'existe).
`boolean isAfterLast()`	Indique si le curseur est positionné après le dernier enregistrement (`false` si aucun enregistrement n'existe).
`void afterLast()`	Positionne le curseur après le dernier enregistrement (aucun effet si le curseur est vide).
`boolean first()`	Positionne le curseur sur le premier enregistrement (`false` si aucun enregistrement n'existe).
`boolean previous()`	Positionne le curseur sur l'enregistrement précédent (`false` si aucun enregistrement ne précède).
`boolean last()`	Positionne le curseur sur le dernier enregistrement (`false` si aucun enregistrement n'existe).
`boolean absolute(int)`	Positionne le curseur sur le n-ième enregistrement (en partant du début si *n* est positif, ou de la fin si *n* est négatif, `false` si aucun enregistrement n'existe à cet indice).
`boolean relative(int)`	Positionne le curseur sur le n-ième enregistrement en partant de la position courante (en avant si *n* est positif, ou en arrière si *n* est négatif, `false` si aucun enregistrement n'existe à cet indice).

Connector/J de MySQL ne permet pas encore de changer le sens de parcours d'un curseur au niveau de l'état et au niveau du curseur lui-même (seule la constante `ResultSet.FETCH_FORWARD` est interprétée). Aucune erreur n'a lieu à l'exécution si vous modifiez le sens de parcours d'un curseur, la direction restera simplement inchangée (en avant toute !).

Ainsi, pour parcourir un curseur à l'envers, il faudra soit utiliser des indices négatifs (dans les méthodes `absolute` et `relative`), soit employer la méthode `previous` en partant de la fin du curseur (`afterLast`).

Parcours

Le code suivant (`SELECTnavigable.java`) présente une utilisation du curseur navigable `curseurNaviJava`. Le deuxième test renvoie `false`, car, après l'ouverture, le curseur n'est pas positionné sur le premier enregistrement, et la méthode `next` le place selon le sens du parcours du curseur.

Tableau 8-20 Parcours d'un curseur navigable

Code Java	Commentaires
`try {…`	
`Statement etatSimple =createStatement (ResultSet.TYPE_SCROLL_INSENSITIVE,ResultSet.CONCUR_READ_ONLY);`	Création de l'état.
`ResultSet curseurNaviJava = etatSimple.executeQuery("SELECT immat,typeAvion,cap FROM Avion");`	Création et chargement du curseur.
`if ( curseurNaviJava.isBeforeFirst() ) System.out.println("Curseur positionné au début");`	Test renvoyant `true`.
`if ( curseurNaviJava.isFirst() ) System.out.println("Curseur positionné sur le 1er déjà");`	Test renvoyant `false`.
`while( curseurNaviJava.next() ) {if ( curseurNaviJava.isFirst() ) System.out.println("1er avion : "); if ( curseurNaviJava.isLast() ) System.out.println("Dernier avion : "); System.out.print("Immat: "+curseurNaviJava.getString(1)); System.out.println(" type : "+curseurNaviJava.getString(2));}`	Parcours du curseur en affichant les premier et dernier enregistrements.
`if ( curseurNaviJava.isAfterLast() )System.out.println("Curseur positionné après la fin");`	Test renvoyant `true`.
`if ( curseurNaviJava.previous() ) if ( curseurNaviJava.previous() ) {System.out.prinln("Avant dernier avion : "+ curseurNaviJava.getString(1));}`	Affiche l'avant-dernier enregistrement.
`if ( curseurNaviJava.first() ) {System.out.println("First avion : "+ curseurNaviJava.getString(1));}`	Affiche le premier enregistrement.

`if ( curseurNaviJava.last()  )`
`  {System.out.println("Last avion : "+`
`  curseurNaviJava.getString(1));}`
	Affiche le dernier enregistrement.

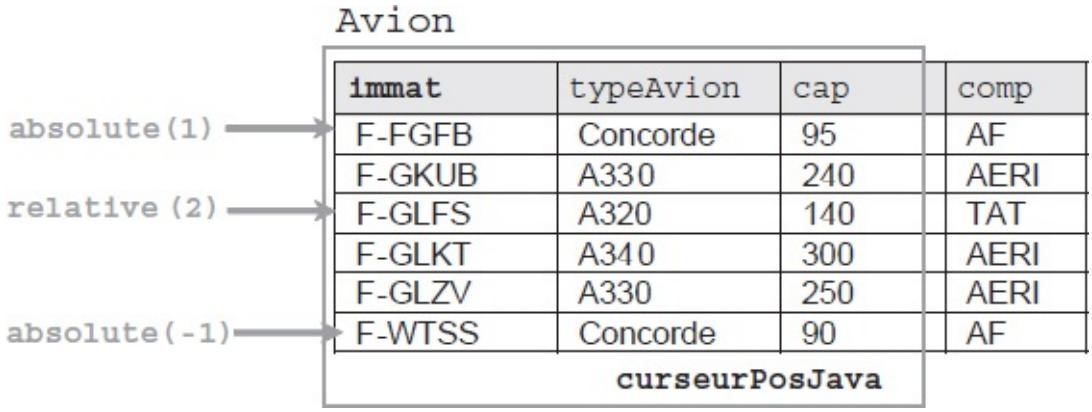
`curseurNaviJava.close();`	Ferme le curseur.
`} catch(SQLException ex) { … }`	Gestion des erreurs.

Créez des curseurs non navigables quand vous voulez rapatrier de très gros volumes de données (taille du cache limitative côté client). Fragmentez vos requêtes quand vous voulez manipuler des curseurs navigables. Les prochaines versions de MySQL et *Connector/J* devraient prendre en charge une gestion côté serveur des curseurs navigables.

Positionnements

Des méthodes assurent l'accès direct à un curseur navigable. Notez que `absolute(1)` équivaut à `first()`, de même `absolute(-1)` équivaut à `last()`. Concernant la méthode `relative`, il faut l'utiliser dans un test pour s'assurer qu'elle s'applique à un enregistrement existant (voir l'exemple suivant). D'autre part, l'utilisation de `relative(0)` n'a aucun effet. Considérons la table suivante qui est interrogée au niveau des trois premières colonnes par le curseur navigable `curseurPosJava` :

Figure 8-5 *Curseur navigable*



Le code suivant (SELECTPositions.java) présente les méthodes qui permettent d'accéder directement à des enregistrements de ce curseur :

Tableau 8-21 Positionnements dans un curseur navigable

Code Java	Commentaires
`try {…`	
`Statement etatSimple =createStatement (ResultSet.TYPE_SCROLL_INSENSITIVE, ResultSet.CONCUR_READ_ONLY);`	Création de l'état avec curseurs insensibles et non modifiables.
`ResultSet curseurPosJava = etatSimple.executeQuery("SELECT immat,typeAvion,cap FROM Avion");`	Création et chargement du curseur.
`curseurPosJava.absolute(1);`	Curseur sur le premier avion.
`if (curseurPosJava.relative(2)) System.out.println("relative(2): "+ curseurPosJava.getString(1));`	Accès au troisième avion.
`else System.out.println("Pas de 3ème avion!");`	
`if (curseurPosJava.relative(-2)) System.out.println("relative(-2) : "+ curseurPosJava.getString(1));`	Retour au premier avion.
`else System.out.println("Pas retour -2 possible !");`	
`if (curseurPosJava.absolute(-2) System.out.println("absolute(-2) : "+ curseurPosJava.getString(1));`	Accès à l'avant-dernier enregistrement.
`else System.out.println("Pas d'avant dernier avion");`	
`curseurPosJava.afterLast(); while(curseurPosJava.previous()) { … }`	Parcours du curseur en sens inverse.
`curseurPosJava.close();`	Fermeture du curseur.
`} catch(SQLException ex) { … }`	Gestion des erreurs.

Pour définir un curseur navigable :

- Une requête ne doit pas contenir de jointure.
- Bien que cela ne soit pas recommandé en production, écrivez « `SELECT a.* FROM table a…` » à la place de « `SELECT * FROM table…` ».

Curseurs modifiables

Un curseur modifiable permet de mettre à jour la base de données : transformation de colonnes, suppressions et insertions d'enregistrements. Les valeurs permises du deuxième paramètre (*`modifCurseur`*) de la méthode `createStatement`, définie à la section précédente, sont présentées dans le tableau suivant :

Tableau 8-22 Constantes de modification d'un curseur

Constante	Explication
`ResultSet.CONCUR_READ_ONLY`	Le curseur ne peut être modifié.
`ResultSet.CONCUR_UPDATABLE`	Le curseur peut être modifié.

Le caractère modifiable d'un curseur est indépendant de sa navigabilité. Néanmoins, il est courant qu'un curseur modifiable soit également navigable (pour pouvoir se positionner à la demande sur un enregistrement avant d'effectuer sa mise à jour).

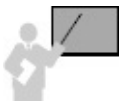
Pour spécifier un curseur de nature `CONCUR_UPDATABLE` :

- Une requête ne doit pas contenir de jointure ni de regroupement, elle doit seulement extraire des colonnes (les fonctions monolignes et multilignes sont interdites).
- Bien que cela ne soit pas recommandé en production, écrivez « `SELECT a.* FROM table a…` » à la place de « `SELECT * FROM table…` » ;

Il est aussi possible de définir un curseur par une requête de type `SELECT… FOR UPDATE`. Les principales méthodes relatives aux curseurs modifiables sont les suivantes :

Tableau 8-23 Méthodes de navigation dans un curseur

Méthode	Fonction
`int getResultSetType()`	Renvoie le caractère navigable des curseurs d'un état donné (`ResultSet.TYPE_FORWARD_ONLY`…).
`int getResultSetConcurrency()`	Renvoie le caractère modifiable des curseurs d'un état donné (`ResultSet.CONCUR_READ_ONLY` ou `ResultSet.CONCUR_UPDATABLE`).
`int getType()`	Renvoie le caractère navigable d'un curseur donné.
`int getConcurrency()`	Renvoie le caractère modifiable d'un curseur donné.
`void deleteRow()`	Supprime l'enregistrement courant.
`void updateRow()`	Modifie la table avec l'enregistrement courant.
`void cancelRowUpdates()`	Annule les modifications faites sur l'enregistrement courant.
`void moveToInsertRow()`	Déplace le curseur vers un nouvel enregistrement.
`void insertRow()`	Insère dans la table l'enregistrement courant.
`void moveToCurrentRow()`	Retour vers l'enregistrement courant (à utiliser éventuellement après `moveToInsertRow`).

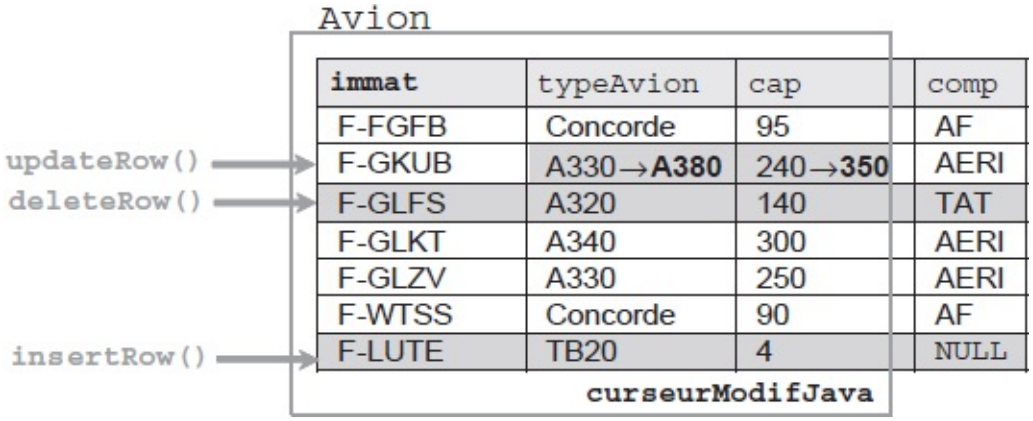


Les opérations de modification et d'insertion (`UPDATE` et `INSERT`) à travers un curseur se réalisent en deux temps : mise à jour du curseur puis propagation à la table de la base de données. Il suffit ainsi de ne pas exécuter la deuxième étape pour ne pas opérer la mise à jour de la base.

La suppression d'enregistrements (`DELETE`) à travers un curseur s'opère en une seule instruction qui n'est pas forcément validée par la suite : il faudra programmer explicitement le `commit` ou laisser le paramètre d'autocommit à `true` (par défaut).

La figure suivante illustre les modifications effectuées sur la table `Avion` par l'intermédiaire du curseur `CurseurModifJava` utilisé par les trois programmes Java suivants :

***Figure 8-6** Mises à jour d'un curseur*



Suppressions

Le code suivant (`ResultDELETE.java`) supprime le troisième enregistrement du curseur et répercute la mise à jour au niveau de la table `Avion` du schéma connecté. Nous déclarons ici ce curseur « navigable » :

Tableau 8-24 Suppression d'un enregistrement

Code Java	Commentaires

`try {…`
`Statement etatSimple =`
`cx.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,`
`ResultSet.CONCUR_UPDATABLE);`
`cx.setAutoCommit(false);`
	Création de l'état et désactivation de la validation automatique.

`ResultSet curseurModifJava = etatSimple.executeQuery` `("SELECT immat,typeAvion,cap FROM Avion");`	Création du curseur.

`if (curseurModifJava.absolute(3))`
`{ curseurModifJava.deleteRow();`
`cx.commit(); }`
`else`
`System.out.println("Pas de 3ème avion!");`
	Accès direct au troisième avion, suppression de l'enregistrement.

`curseurModifJava.close();`	Fermeture du curseur.

`} catch(SQLException ex) { … }`	Gestion des erreurs.

Le code suivant (`ResultDELETE2.java`) supprime le même enregistrement en supposant son indice a priori inconnu. Nous déclarons ici ce curseur « non navigable ». Notez l'utilisation de la méthode `equals` pour comparer deux chaînes de caractères :

Tableau 8-25 Suppression d'un enregistrement

Code Java	Commentaires

`try {…`
`Statement etatSimple =`
`cx.createStatement(ResultSet.TYPE_FORWARD_ONLY,`
`ResultSet.CONCUR_UPDATABLE);`
`cx.setAutoCommit(false);`
	Création de l'état et désactivation de la validation automatique.

`ResultSet curseurModifJava = etatSimple.executeQuery` `("SELECT immat,typeAvion,cap FROM Avion");`	Création du curseur.
`String p_immat = "F-GLFS";`	

`while(curseurModifJava.next())`
`{if (curseurModifJava.getString(1).equals(p_immat))`
`{curseurModifJava.deleteRow();; }`
`}`
	Accès à l'enregistrement et suppression.

`curseurModifJava.close();`	Fermeture du curseur.
`} catch(SQLException ex) { … }`	Gestion des erreurs.

Modifications

La modification de colonnes d'un enregistrement au niveau de la base de données s'opère en deux étapes : mise à jour du curseur par les méthodes `updatexxx` (*updater methods*), puis propagation des mises à jour dans la table par la méthode `updateRow()`.

Les méthodes `updatexxx` ont chacune deux signatures. Par exemple, la méthode de modification d'une chaîne de caractères (valable pour les colonnes `CHAR` et `VARCHAR`) est disponible en raisonnant en fonction soit de la position soit du nom de la colonne du curseur :

```
void updateString(int positionColonne, String chaîne)
```

```
void updateString(String nomColonne, String chaîne)
```

Le code suivant (ResultUPDATE.java) change, au niveau de la table Avion, deux colonnes du deuxième enregistrement du curseur. Nous déclarons ici ce curseur « sensible » pour pouvoir éventuellement visualiser la transformation réalisée dans le même programme.

Tableau 8-26 Modifications d'un enregistrement

Code Java	Commentaires
`try {…`	
`Statement etatSimple =`	
`cx.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,`	
`ResultSet.CONCUR_UPDATABLE);`	Création de l'état et désactivation de la validation automatique.
`cx.setAutoCommit(false);`	
`ResultSet curseurModifJava = etatSimple.executeQuery`	
`("SELECT immat,typeAvion,cap FROM Avion");`	Création du curseur.
`if (curseurModifJava.absolute(2))`	Accès à l'enregistrement.
`{ curseurModifJava.updateString(2,"A380");`	Première étape.
`curseurModifJava.updateInt(3,350);`	
`curseurModifJava.updateRow();`	Deuxième étape.
`cx.commit(); }`	
`else`	
`System.out.println("Pas de 2ème avion!");`	Validation.
`curseurModifJava.close();`	Fermeture du curseur.
`} catch(SQLException ex) { … }`	Gestion des erreurs.

Insertions

L'insertion d'un enregistrement au niveau de la base de données s'opère en trois étapes : préparation à l'insertion dans le curseur par la méthode moveToInsertRow, mise à jour du curseur par les méthodes updatexxx, puis propagation des actualisations dans la table par la méthode insertRow. L'éventuel retour à l'enregistrement courant se programme à l'aide de la méthode moveToCurrentRow.

Le code suivant (ResultINSERT.java) insère un nouvel enregistrement au niveau de la table Avion. La quatrième colonne de la table n'est pas indiquée dans le

curseur, elle est donc passée à NULL au niveau de la table, en l'absence de valeur par défaut définie dans la colonne.

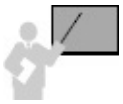
Tableau 8-27 Insertion d'un enregistrement

Code Java	Commentaires
`try {…`	
`Statement etatSimple =`	
`cx.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,`	
`ResultSet.CONCUR_UPDATABLE);`	Création de l'état et désactivation de la validation automatique.
`cx.setAutoCommit(false);`	
`ResultSet curseurModifJava = etatSimple.executeQuery`	
`("SELECT immat,typeAvion,cap FROM Avion");`	Création du curseur.
`curseurModifJava.moveToInsertRow();`	Première étape.
`curseurModifJava.updateString(1,"F-LUTE");`	
`curseurModifJava.updateString(2,"TB20");`	Deuxième étape.
`curseurModifJava.updateInt(3,4);`	
`curseurModifJava.insertRow();`	
`cx.commit();`	Troisième étape. Validation.
`curseurModifJava.close();`	Fermeture du curseur.
`} catch(SQLException ex) { … }`	Gestion des erreurs.

Gestion des séquences

Avant la version 3.0 de l'API JDBC de Sun, il n'y avait pas de possibilité « standard » d'extraire la valeur courante d'une séquence (colonne AUTO_INCREMENT). Avec des anciens pilotes JDBC pour MySQL, il était possible d'utiliser une méthode spécifique de l'interface Statement, ou d'exécuter une requête du type SELECT LAST_INSERT_ID() après avoir inséré un enregistrement. Le premier mécanisme n'assure pas la portabilité, le second n'est pas efficace, puisqu'il oblige à insérer une ligne au préalable (qu'on peut toutefois annuler avec un rollback).

À présent, JDBC 3.0 offre deux nouveaux mécanismes afin d'extraire la valeur courante d'une séquence AUTO_INCREMENT : la méthode getGeneratedKeys() ou l'exploitation d'un curseur modifiable via la méthode insertRow().



Dans tous les cas, il n'est pas nécessaire d'insérer réellement un nouvel enregistrement (annulation possible par `rollback`). En revanche, l'exécution de l'ajout (par `INSERT`) est obligatoire pour que MySQL évalue une nouvelle valeur de la séquence. Cette action est irréversible dans le sens où la séquence sera incrémentée qu'on valide ou non l'ajout de l'enregistrement.

Méthode `getGeneratedKeys`

Le code suivant (`Sequence1.java`) insère un nouvel enregistrement dans la table `Affreter` (décrite au chapitre 2, section « Séquences ») et récupère la valeur courante de la séquence à l'aide de la méthode `getGeneratedKeys()`.

Tableau 8-28 Récupération d'une séquence à l'aide d'un état

Code Java	Commentaires

`try {…`
`Statement etat = cx.createStatement`
`(java.sql.ResultSet.TYPE_FORWARD_ONLY,`
`java.sql.ResultSet.CONCUR_UPDATABLE);`
`cx.setAutoCommit(false);`
	Création de l'état et désactivation de la validation automatique.

`etat.execute("INSERT INTO bdsoutou.Affreter (comp,immat,dateAff,nbPax)VALUES('AF','A1',NOW(),100)",` `Statement.RETURN_GENERATED_KEYS);`	Insertion avec l'option de récupération de clé générée.

`ResultSet curseur = etat.getGeneratedKeys();`
`if (curseur.next())`
`System.out.println("Valeur de la sequence "+`
`curseur.getInt(1));`
`else System.out.println("Pb sequence ");`
`cx.rollback();`
	Extraction de la séquence.

`curseur.close(); etat.close(); cx.close();`	Invalidation de l'ajout. Fermeture des objets.
`} catch(SQLException ex) { … }`	Gestion des erreurs.

Curseur modifiable

Le code suivant (`Sequence2.java`) insère un nouvel enregistrement dans la table `Affreter` par un curseur modifiable, et récupère la valeur courante de la séquence à l'aide de la méthode `getInt()` appliquée à la colonne `AUTO_INCREMENT`.

Tableau 8-29 Récupération d'une séquence à l'aide d'un curseur modifiable

Code Java	Commentaires

`try {…`
`Statement etat = cx.createStatement`
`(java.sql.ResultSet.TYPE_FORWARD_ONLY,`
`java.sql.ResultSet.CONCUR_UPDATABLE);`
`cx.setAutoCommit(false);`
	Création de l'état et désactivation de la validation automatique.

`ResultSet curseur = etat.executeQuery("SELECT numAff,immat,comp,dateAff FROM bdsoutou.Affreter");`	
`curseur.moveToInsertRow();` `curseur.insertRow();`	Insertion via le curseur.

`curseur.last();`
`System.out.println("Valeur de la sequence "+`
`curseur.getInt(1));`
	Extraction de la séquence.

`cx.rollback();`	Invalidation de l'ajout.
`curseur.close(); etat.close(); cx.close();`	Fermeture des objets.
`} catch(SQLException ex) { … }`	Gestion des erreurs.

Interface `ResultSetMetaData`

L'interface `ResultSetMetaData` est utile pour retrouver dynamiquement des propriétés des tables qui sont manipulées par des curseurs `ResultSet`. Cette interface est intéressante pour programmer dynamiquement des requêtes ou d'autres instructions SQL. Ces fonctions vont extraire de manière transparente des informations par l'intermédiaire du dictionnaire des données.

Une fois un curseur `ResultSet` programmé, il suffit de lui appliquer la méthode `getMetaData()` pour disposer d'un objet `ResultSetMetaData`. Le tableau suivant présente les principales méthodes disponibles de l'interface `ResultSetMetaData` :

Tableau 8-30 Méthodes principales de l'interface ResultSetMetaData

Méthode	Description
`int getColumnCount()`	Retourne le nombre de colonnes du curseur.
`String getColumnName(int)`	Retourne le nom de la colonne d'un indice donné du curseur.
`int getColumnType(int)`	Retourne le code du type (selon la classification de `java.sql.Types`) de la colonne d'un indice donné du curseur.
`String getColumnTypeName(int)`	Retourne le nom du type SQL de la colonne d'un indice donné du curseur.
`int isNullable(int)`	Indique si la colonne d'un indice donné du curseur peut être nulle (constantes retournées : `ResultSetMetaData.columnNoNulls`, `ResultSetMetaData.columnNullable` ou `ResultSetMetaData.columnNullableUnknown`).
`int getPrecision(int)`	Nombre de chiffres avant la virgule de la colonne désignée.
`int getScale(int)`	Nombre de décimales de la colonne désignée.
`String getSchemaName(int)`	Nom du schéma propriétaire de la colonne.
`String getTableName(int)`	Nom de la table de la colonne.

Comme nous l'avons vu au chapitre 5, MySQL ne renseigne pas encore le nom de catalogue. Ainsi, la méthode `getSchemaName`() n'est pas encore reconnue.

Le code suivant (`ResulSetMeta.java`) utilise des méthodes de l'interface `ResultSetMetaData` sur la base de la requête extrayant trois colonnes dans la table `Avion` :

Tableau 8-31 Extraction de méta-informations au niveau d'un curseur

Code Java	Commentaires
`try { …`	

`ResultSet curseurJava=etatSimple.executeQuery ("SELECT immat, typeAvion, cap FROM Avion");`	Création du curseur.
`ResultSetMetaData rsmd = curseurJava.getMetaData();`	Création d'un objet `ResultSetMetaData`.
`int nbCol = rsmd.getColumnCount();`	`nbCol` contient 3.
`String nom2emeCol = rsmd.getColumnName(2);`	`nom2emeCol` contient `typeAvion`.
`String type2emeCol = rsmd.getColumnTypeName(2);`	`type2emeCol` contient `VARCHAR`.
`int codeType2emeCol = rsmd.getColumnType(2);`	`codeType2emeCol` contient 12 (code pour `VARCHAR`).
`if (rsmd.isNullable(1) == ResultSetMetaData.columnNoNulls) …`	Test renvoyant vrai (la première colonne est la clé primaire).
`int p1 = rsmd.getPrecision(3);`	Taille de la colonne `cap` (renvoie 6 pour un `SMALLINT`).
`int t1 = rsmd.getScale(3);`	Décimales de la colonne `cap` (renvoie 0 pour un `SMALLINT`).
`curseurJava.close();`	Fermeture du curseur.
`} catch(SQLException ex) { … }`	Gestion des erreurs.

Interface `DatabaseMetaData`

L'interface `DatabaseMetaData` est utile pour connaître des aspects plus généraux de la base de données cible (version, éditeur, prise en charge des transactions…) ou des informations sur la structure de la base (structures des tables et vues, prérogatives…).

Plus de quarante méthodes sont proposées par l'interface `DatabaseMetaData`. Le tableau suivant en présente quelques-unes. Consultez la documentation du JDK pour en savoir plus.

Tableau 8-32 Méthodes principales de l'interface ResultSetMetaData

Méthode	Description
`ResultSet getColumns(String, String, String, String)`	Description de toutes les colonnes d'une table d'un schéma donné.
`String getDatabaseProductName()`	Nom de l'éditeur de la base de données utilisée.

`String getDatabaseProductVersion()`	Numéro de la version de la base utilisée.
`ResultSet getTables(String, String, String, String[])`	Description des tables d'un schéma donné.
`String getUserName()`	Nom de l'utilisateur connecté (schéma courant).
`boolean supportsSavepoints()`	Renvoie `true` si la base reconnaît les points de validation.
`boolean supportsTransactions()`	Renvoie `true` si la base reconnaît les transactions.

Le code suivant (`MetaData.java`) emploie ces méthodes pour extraire des informations à propos de la base cible et des objets (tables, vues, séquences…) du schéma courant.

Tableau 8-33 Extraction de méta-informations au niveau d'un schéma

Code Java	Commentaires
`try {` ` …`	

` DatabaseMetaData infoBase = cx.getMetaData();`
 	
` ResultSet toutesLesTables = infoBase.getTables("",`
`  infoBase.getUserName(), null, null);`
	Création d'un objet `DatabaseMetaData`.

`System.out.println("Objets du schema "+` `  infoBase.getUserName());`	Création d'un objet `ResultSet` contenant les caractéristiques du schéma courant.

`while (toutesLesTables.next())`
` {  System.out.print("Nom de l'objet: "+`
`  toutesLesTables.getString(3));`
`  System.out.println(" Type : "+`
`  toutesLesTables.getString(4)); }`
	Parcours du curseur en affichant quelques caractéristiques.

`System.out.println("Nom base : "+` `  infoBase.getDatabaseProductName());`	Affichage du nom de la base.
`  System.out.println("Version base : "+` `  infoBase.getDatabaseProductVersion ());`	Affichage de la version de la base.
`if (infoBase.supportsTransactions())` `  System.out.println("Supporte les Transactions");`	Transactions prises en charge ou pas.

`toutesLesTables.close();`	Fermeture du curseur.
`} catch(SQLException ex) { … }`	Gestion des erreurs.

La trace de ce programme est la suivante (dans notre jeu d'exemples) :

```
Objets du schema soutou@localhost
Nom de l'objet: Avion Type : TABLE
Nom de l'objet: Compagnie Type : TABLE
Nom base : MySQL
Version base : 5.0.15-nt
Supporte les SelectForUpdate
Supporte les Transactions
```

Instructions paramétrées [`PreparedStatement`]

L'interface `PreparedStatement` hérite de l'interface `Statement`, et la spécialise en permettant de paramétrer des objets (états préparés) représentant des instructions SQL précompilées. Ces états sont créés par la méthode `prepareStatement` de l'interface `Connection` décrite ci-après. La chaîne de caractères contient l'ordre SQL dont les paramètres, s'il en possède, doivent être indiqués par le symbole « ? ».

```
PreparedStatement prepareStatement(String)
```

Une fois créés, ces objets peuvent être aisément réutilisés pour exécuter à la demande l'instruction SQL, en modifiant éventuellement les valeurs des paramètres d'entrée à l'aide des méthodes `setxxx` (*setter methods*). Le tableau suivant décrit les principales méthodes de l'interface `preparedStatement` :

Tableau 8-34 Méthodes de l'interface PreparedStatement

Méthode	Description
`ResultSet executeQuery()`	Exécute la requête et retourne un curseur ni navigable, ni modifiable par défaut.
`int executeUpdate()`	Exécute une instruction LMD (`INSERT`, `UPDATE` ou `DELETE`) et retourne le nombre de lignes traitées, ou 0 pour les instructions SQL ne retournant aucun résultat (LDD).
`boolean execute()`	Exécute une instruction SQL et renvoie `true`,

	si c'est une instruction `SELECT`, `false` sinon.
`void setNull(int, int)`	Affecte la valeur `NULL` au paramètre de numéro et de type (classification `java.sql.Types`) spécifiés.
`void close()`	Ferme l'état.

Décrivons à présent un exemple d'appel pour chaque méthode de compilation d'un ordre paramétré. On suppose la connexion `cx` créée :

Extraction de données [executeQuery]

Le code suivant (`PrepareSELECT.java`) illustre l'utilisation de la méthode `executeQuery` pour extraire les enregistrements de la table `Avion` :

Tableau 8-35 Extraction de données par un ordre préparé

Code Java	Commentaires
`try { …`	
`String ordreSQL =`	
`"SELECT immat, typeAvion, cap FROM Avion";`	
`PreparedStatement étatPréparé =`	Création d'un état préparé.
`cx.prepareStatement(ordreSQL);`	
`ResultSet curseurJava =étatPréparé.executeQuery();`	Création du curseur résultant de la compilation de l'état.
`while(curseurJava.next())`	
`{ … }`	Parcours du curseur.
`curseurJava.close();`	Fermeture du curseur.
`étatPréparé.close();`	Fermeture de l'état.
`} catch(SQLException ex) { … }`	Gestion des erreurs.

Mises à jour [executeUpdate]

Le code suivant (`PrepareINSERT.java`) illustre l'utilisation de la méthode `executeUpdate` pour insérer l'enregistrement (`F-NEW`, `A319`, `178`, `AF`) dans la table `Avion` composée de quatre colonnes de types `CHAR(6)`, `VARCHAR(15)`, `SMALLINT` et `VARCHAR(4)` :

Tableau 8-36 Insertion d'un enregistrement par un ordre préparé

Code Java	Commentaires

`try { …`
`String ordreSQL =`
`"INSERT INTO Avion VALUES (?, ?, ?, ?)";`
`PreparedStatement étatPréparé =`
`cx.prepareStatement(ordreSQL);`
	Création d'un état préparé.

`étatPréparé.setString(1, "F-NEW");`	Passage des paramètres.
`étatPréparé.setString(2, "A319");`	
`étatPréparé.setInt(3, 178);`	
`étatPréparé.setString(4, "AF");`	
`System.out.println(étatPréparé.executeUpdate()` `+ " avion inséré.");`	Exécution de l'instruction.
`étatPréparé.close();`	Fermeture de l'état.
`} catch(SQLException ex) { … }`	Gestion des erreurs.

Instruction LDD [execute]

Le code suivant (PrepareDELETE.java) illustre l'utilisation de la méthode execute pour supprimer un avion dont l'immatriculation passe en paramètre :

Tableau 8-37 Insertion d'un enregistrement par un ordre préparé

Code Java	Commentaires

`try { …`
`cx.setAutoCommit(false);`
`String ordreSQL =`
`"DELETE FROM Avion WHERE immat = ?";`
`PreparedStatement étatPréparé =`	

`cx.prepareStatement(ordreSQL);`
`étatPréparé.setString(1, "F-NEW ");`
`if (! étatPréparé.execute() )`
	Création d'un état préparé.
	Passage du paramètre.
	Exécution de l'instruction.

`{ System.out.println("Enregistrement supprimé");` `cx.commit(); }`	
`étatPréparé.close();`	Fermeture de l'état.
`} catch(SQLException ex) { … }`	Gestion des erreurs.

Il n'est pas possible de paramétrer des instructions SQL du LDD (CREATE, ALTER…). Pour résoudre ce problème, il faut construire dynamiquement la chaîne (String) qui contient l'instruction à l'aide de l'opérateur de

concaténation Java (+). Cette chaîne sera ensuite l'unique paramètre de la méthode `prepareStatement`.

Procédures cataloguées

L'interface `CallableStatement` permet d'appeler des sous-programmes (fonctions ou procédures cataloguées), en passant d'éventuels paramètres en entrée et en en récupérant en sortie. L'interface `CallableStatement` spécialise l'interface `PreparedStatement`. Les paramètres d'entrée sont affectés par les méthodes `setxxx`. Les paramètres de sortie (définis `OUT` au niveau du sous-programme) sont extraits à l'aide des méthodes `getxxx`. Ces états qui permettent d'appeler des sous-programmes sont créés par la méthode `prepareCall` de l'interface `Connection`, décrite ci-après :

```
CallableStatement prepareCall(String)
```

Le tableau suivant décompose le paramètre de cette méthode (deux écritures sont possibles). Chaque paramètre est indiqué par un symbole « ? » :

Tableau 8-38 Paramètre de prepareCall

Type du sous-programme	Paramètre
Fonction	`{? = call nomFonction( [?, ?, ...] ) }`
Procédure	`{call nomProcédure( [?, ?, ...] ) }`

Une fois l'état créé, il faut répertorier le type des paramètres de sortie (méthode `registerOutParameter`), passer les valeurs des paramètres d'entrée, appeler le sous-programme et analyser les résultats. Le tableau suivant décrit les principales méthodes de l'interface `CallableStatement` :

Tableau 8-39 Méthodes de l'interface CallableStatement

Méthode	Description
`ResultSet executeQuery()`	Idem `PreparedStatement`.
`int executeUpdate()`	Idem `PreparedStatement`.
`boolean execute()`	Idem `PreparedStatement`.
`void registerOutParameter(int, int)`	Transfère un paramètre de sortie à un indice donné d'un type Java

	(classification `java.sql.Types`).
`boolean wasNull()`	Détermine si le dernier paramètre de sortie extrait est à `NULL`. Cette méthode doit être seulement invoquée après une méthode de type `get`*XXX*.

Exemple

Le programme JDBC suivant (`CallableProcedure.java`) décrit l'appel de la procédure `leNomCompagnieEst` (ayant deux paramètres). Le premier indique l'avion de la compagnie recherché, le second contient le résultat (nom de la compagnie).

```
CREATE PROCEDURE bdsoutou.leNomCompagnieEst
      (IN p_immat CHAR(6),OUT p_nomcomp VARCHAR(25))
BEGIN
 DECLARE flagNOTFOUND BOOLEAN DEFAULT 0;
 BEGIN
   DECLARE EXIT HANDLER FOR NOT FOUND SET flagNOTFOUND :=1;
   SELECT nomComp INTO p_nomcomp FROM Compagnie
   WHERE comp = (SELECT compa FROM Avion WHERE immat = p_immat);
 END;
 IF flagNOTFOUND THEN
    SET p_nomcomp := NULL;
 END IF;
END;
```

Le tableau suivant décrit les étapes nécessaires à l'appel de cette procédure (qui ne gère pas les éventuelles erreurs) pour l'avion d'immatriculation 'F-GLFS'.

Tableau 8-40 Appel d'une procédure (paramètre en entrée et sortie)

Code Java	Commentaires
`String ordreSQL = " {call bdsoutou.leNomCompagnieEst(?,?)} ";`	
`CallableStatement étatAppelable = cx.prepareCall(ordreSQL);`	Création d'un état appelable.
`étatAppelable.registerOutParameter (2,java.sql.Types.VARCHAR);`	Déclaration du paramètre de sortie.
`étatAppelable.setString(1,"F-GLFS");`	Passage du paramètre d'entrée.
`étatAppelable.execute();`	Exécution de la procédure.

```
  System.out.print("Compagnie de F-GLFS : "+
                   étatAppelable.getString(2));

  étatAppelable.close();

} catch(SQLException ex) { … }
```

Extraction du résultat.
Fermeture de l'état.
Gestion des erreurs.

La trace de l'appel de cette procédure est la suivante :

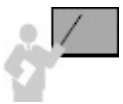
```
Compagnie de F-GLFS : Transport Air Tour
```

Pensez, sous *root*, à donner les privilèges nécessaires à l'utilisateur qui va lancer le sous-programme via Java (pour mon test, j'ai dû lancer `GRANT EXECUTE ON bdsoutou.* TO 'soutou'@'localhost'` et `GRANT SELECT ON mysql.proc TO 'soutou'@'localhost'`). Si vous ne le faites pas, JDBC vous rappellera clairement à l'ordre.

Transactions

JDBC supporte le mode transactionnel qui consiste à valider tout ou une partie d'un ensemble d'instructions. Nous avons déjà décrit, à la section « Interface Connection », les méthodes qui permettent à un programme Java de coder des transactions (`setAutoCommit`, `commit` et `rollback`).

Par défaut, chaque instruction SQL est validée (on parle *d'autocommit*). Lorsque ce mode est désactivé, il faut gérer manuellement les transactions avec `commit` ou `rollback`.



Quand le mode *autocommit* est désactivé :

- La déconnexion d'un objet `Connection` (par la méthode `close`) valide implicitement la transaction (même si `commit` n'a pas été invoqué avant la déconnexion).
- Chaque instruction du LDD (`CREATE`, `ALTER`, `DROP`) valide implicitement la transaction.

Points de validation

Depuis la version 3.0 de JDBC (JDK 1.4), on peut inclure des points de validation et affiner ainsi la programmation des transactions. Les interfaces `Connection` et `Savepoint` rendent possible cette programmation.

Interface `Connection`

Le tableau suivant présente les méthodes de l'interface `Connection` qui sont relatives au principe des points de validation :

Tableau 8-41 Méthodes concernant les points de validation de l'interface Connection

Méthode	Description
`Savepoint setSavepoint()`	Positionne un point de validation anonyme et retourne un objet `Savepoint`.
`Savepoint setSavepoint(String)`	Positionne un point de validation nommé et retourne un objet `Savepoint`.
`void releaseSavepoint(Savepoint)`	Supprime le point de validation de la transaction courante.
`void rollback(Savepoint)`	Invalide la transaction à partir du point de validation.

Interface `Savepoint`

Les points de validation sont anonymes (identifiés toutefois par un entier) ou nommés. Le tableau suivant présente les deux seules méthodes de l'interface `Savepoint` :

Tableau 8-42 Méthodes de l'interface Savepoint

Méthode	Description
`int getSavepointId()`	Retourne l'identifiant du point de validation de l'objet `Savepoint`.
`String getSavepointName()`	Retourne le nom du point de validation de l'objet `Savepoint`.

Le code suivant (`TransactionJDBC.java`) illustre une transaction découpée en deux phases par deux points de validation. Dans notre exemple, nous validons seulement la première partie (seul l'avion '`F-NEW2`' sera inséré dans la table). On suppose la connexion `cx` créée.

Tableau 8-43 Points de validation

Code Java	Commentaires
`try { …`	
`cx.setAutoCommit(false);`	Désactivation de *l'autocommit*.
`String ordreSQL =`	
`"INSERT INTO Avion VALUES (?, ?, ?, ?)";`	
`PreparedStatement étatPréparé =`	
`cx.prepareStatement(ordreSQL);`	
`Savepoint p1 = cx.setSavepoint("P1");`	Création d'un état appelable. Création du point de validation P1.
`étatPréparé.setString(1, "F-NEW2");`	Passage de paramètres et première insertion.
`…`	
`if (! étatPréparé.execute() )`	
`System.out.println("F-NEW2 inséré");`	
`Savepoint p2 = cx.setSavepoint("P2");`	Création du point de validation P2.
`étatPréparé.setString(1, "F-NEW3");`	Passage de paramètres et deuxième insertion.
`…`	
`if (! étatPréparé.execute() )`	
`System.out.println("F-NEW3 inséré");`	
`cx.rollback(p2);`	Annulation de la deuxième partie.
`cx.commit();`	Validation de la première partie.
`cx.close();`	Fermeture de la connexion.
`} catch(SQLException ex) { … }`	Gestion des erreurs.

Traitement des exceptions

Les exceptions qui ne sont pas traitées dans les sous-programmes appelés, ou celles que les sous-programmes ou déclencheurs peuvent retourner, doivent être prises en compte au niveau du code Java (dans un bloc `try`… `catch`…). Le bloc d'exceptions permet de programmer des traitements en fonction des codes d'erreur renvoyés par la base MySQL. Plusieurs blocs d'exceptions peuvent être imbriqués dans un programme JDBC.

Afin de gérer les erreurs renvoyées par le SGBD, JDBC propose la classe `SQLException` qui hérite de la classe `Exception`. Chaque objet (automatiquement créé dès la première erreur) de cette classe dispose des méthodes suivantes :

Tableau 8-44 Méthodes de la classe SQLException

Méthode	Description
`String getMessage()`	Message décrivant l'erreur.
`String getSQLState()`	Code erreur SQL Standard (XOPEN ou SQL99).
`int getErrorCode()`	Code erreur SQL de la base.
`SQLException getNextException()`	Chaînage à l'exception suivante (si une erreur renvoie plusieurs messages).

Affichage des erreurs

Le code suivant (`Exceptions1.java`) illustre une manière d'afficher explicitement toutes les erreurs sans effectuer d'autres instructions :

Tableau 8-45 Affichage des erreurs

Code Java	Commentaires

`import java.sql.*;`
`public class Exceptions1`
`{ public static void main`
`(String args []) throws SQLException, Exception`
`{try`
`{Class.forName`
`("com.mysql.jdbc.Driver").newInstance();}`
`catch (ClassNotFoundException ex)`
`{System.out.println`
`("Problème au chargement"+ex.toString()); }`
`try`
`{ Connection cx =`
`DriverManager.getConnection(...`
`cx.close(); }`
`catch(SQLException ex)`
`{System.err.println("Erreur");`
`while ((ex != null))`
`{System.err.println("Statut : "+ ex.getSQLState());`
`System.err.println("Message : "+ ex.getMessage());`
`System.err.println("Code base : "+ex.getErrorCode());`
`ex = ex.getNextException();}`
`} } }
	Classe principale.
	Chargement du pilote.
	Connexion.
	Instructions…
	Gestion des erreurs.

Traitement des erreurs

Il est possible d'associer des traitements à chaque erreur répertoriée avant l'exécution du programme. On peut appeler des méthodes de la classe principale ou coder directement dans le bloc des exceptions.

Le code suivant (`Exceptions2.java`) insère un enregistrement dans la table `Avion` en gérant un certain nombre d'exceptions possibles. Le premier bloc des exceptions permet d'afficher un message personnalisé pour chaque type d'erreur préalablement répertorié (duplication de clé primaire, mauvais nombre ou type de colonnes…). Si l'avion à insérer n'est pas rattaché à une compagnie existante (contrainte référentielle), exception *`1452-Cannot add or update a child row: a foreign key constraint fails`*). Le dernier bloc d'exceptions affiche une erreur qui n'a pas été prévue par le programmeur (erreur système ou de syntaxe dans l'instruction, par exemple).

Tableau 8-46 Traitement des exceptions

Code Java	Commentaires

`try`
`{Connection cx =`
`DriverManager.getConnection(…);`
`String ordreSQL =`
`"INSERT INTO bdsoutou.Avion VALUES`
`('F-NOUV','A310', 5600, 'AF')";`
`PreparedStatement étatPréparé =`
`cx.prepareStatement(ordreSQL);`
`System.out.println(étatPréparé.executeUpdate() +`
`" avion insere en base.");`
`cx.close(); }`
	Instructions du `main`.

`catch(SQLException ex){`	Non unique.
`if (ex.getErrorCode() == 1062)` `System.out.println("Avion déjà existant!");`	
`else if (ex.getErrorCode() == 1044)` `System.out.println("Nom de base inconnu!");`	Mauvais nom de base.
`else if (ex.getErrorCode() == 1136)` `System.out.println("Trop ou pas assez de valeurs!");`	Mauvais nombre de colonnes.
`else if (ex.getErrorCode() == 1146)` `System.out.println("Nom de table inconnue!");`	Mauvais nom de table.
`else if ((ex.getSQLState() == "01004") &&` `(ex.getErrorCode() == 0))`	Mauvais type de colonnes.
`System.out.println("Valeur trop longue ou` `valeur trop importante!");`	

```
else if (ex.getErrorCode() == 1452)
   System.out.println("Compagnie inconnue, a
              inserer avec l'avion");
else
  System.err.println("Erreur");
  while ((ex != null))
  {System.err.println("Statut : "+ ex.getSQLState());
   System.err.println("Message : "+ ex.getMessage());
   System.err.println("Code Erreur base : "+
            ex.getErrorCode());
   ex = ex.getNextException();} }
```

Clé étrangère absente.

Gestion des autres erreurs.

Exercices

L'objectif de ces exercices est de développer des méthodes de la classe Java `ExoJDBC` pour extraire et mettre à jour des données des tables du schéma *Parc informatique*.

Exercice 8.1

Curseur statique

Écrire les méthodes :

• `ArrayList getSalles()` qui retourne sous la forme d'une liste les enregistrements de la table `Salle`.

• `main` qui se connecte à la base, appelle la méthode `getSalles` et affiche les résultats (exemple donné ci-dessous) :

```
nSalle  nomSalle        nbPoste indIP
----------------------------------------
s01     Salle 1         3       130.120.80
s02     Salle 2         2       130.120.80
…
```

Ajoutez une nouvelle salle dans la table `Salle` dans l'interface de commande, et lancez à nouveau le programme pour vérifier.

Exercice 8.2

Curseur modifiable

Écrivez la méthode `void deleteSalle(int)` qui supprime de la table `Salle` l'enregistrement de rang passé en paramètre. Vous utiliserez la méthode `deleteRow` appliquée à un curseur modifiable. Appelez dans le `main` cette méthode

pour supprimer l'enregistrement de la table `Salle` que vous avez ajouté en test, dans l'exercice précédent. Si l'enregistrement est rattaché à un enregistrement *fils*, ne forcez pas la contrainte référentielle, contentez-vous d'afficher le message d'erreur 1451 renvoyé par MySQL, dans le bloc des exceptions.

Exercice 8.3

Appel d'un sous-programme

Compiler, dans votre base, la procédure cataloguée `supprimeSalle(IN ns VARCHAR(7),OUT res TINYINT)` qui supprime une salle dont le numéro est passé en premier paramètre.

```
CREATE PROCEDURE supprimeSalle(IN ns VARCHAR(7),OUT res TINYINT)
BEGIN
 DECLARE ligne VARCHAR(20);
 DECLARE EXIT HANDLER FOR NOT FOUND    SET res := -1;
 DECLARE EXIT HANDLER FOR SQLEXCEPTION SET res := -2;
 SELECT nomSalle INTO ligne FROM Salle WHERE nSalle = ns;
 DELETE FROM Salle WHERE nsalle = ns;
 SET res := 0;
 COMMIT ;
END;
```

La procédure retourne en second paramètre :

• 0 si la suppression s'est déroulée correctement ;

• -1 si le code de la salle est inconnu ;

• -2 si la suppression est impossible (contraintes référentielles).

Écrire la méthode Java `int deleteSalleSP(String)` qui appelle le sous-programme `supprimeSalle`. Essayer les différents cas d'erreurs en appelant cette méthode d'abord avec un numéro de salle référencé par un poste de travail, et ensuite avec un numéro de salle inexistant. Penser à donner à l'utilisateur le privilège en exécution sur cette procédure.

Chapitre 9

Utilisation avec PHP

Ce chapitre explique les moyens de faire interagir un programme PHP 5 avec une base MySQL.

Depuis sa version 5.1, le framework PHP propose trois extensions pour MySQL (comparable à ce que MySQL appelle *connector* et *driver*) :

- `mysql_` qui convient à des bases MySQL de version antérieure à 4.1 (documentation disponible sur http://www.php.net/manual/fr/book.mysql.php).
- `mysqli` qui consiste en une version améliorée de la précédente (`i` comme *improved*) et qui convient à des bases MySQL de version postérieure à 4.1 tout en supportant les plus anciennes jusqu'à la version 3.22 (documentation disponible sur http://www.php.net/manual/fr/book.mysqli.php).
- PDO (*PHP Data Objects*), couche d'abstraction orientée objet qui peut s'adapter à d'autres bases de données par l'intermédiaire d'un pilote (documentation disponible sur http://php.net/manual/fr/ref.pdo-mysql.php).

L'extension `mysqli` dispose de nombreux atouts : elle supporte toutes les commandes de MySQL, même les plus récentes, les transactions, le débogage et le serveur embarqué. En plus d'une interface procédurale, `mysqli` propose aussi un style d'écriture orienté objet (abordé en fin de chapitre).

Configuration adoptée

Plusieurs configurations sont possibles pour intégrer PHP et MySQL. Dans la majorité des cas, un serveur HTTP est utilisé (le plus souvent il s'agit d'Apache). Sous Windows, la solution centralisée WampServer (http://www.wampserver.com) inclus Apache, MySQL et PHP. Sous GNU/Linux, des paquetages LAMP existent aussi.

En installant chaque programme séparément, il est aussi possible de faire interagir une base MySQL avec un programme PHP. Lorsque la configuration de test a été installée, il s'agissait de PHP 5.5.3, d'une base MySQL 5.5.6 et d'Apache 2.2.16. La procédure d'installation sous Windows décrite ici est minimale, vous trouverez sur le Web de nombreuses ressources à ce sujet.

Logiciels

Téléchargez, puis installez Apache (http://httpd.apache.org/download.cgi). Testez le serveur dans le navigateur (`http://localhost` dans mon cas).

Téléchargez, puis installez PHP (http://www.php.net/downloads.php) en décompressant le fichier téléchargé dans un de vos répertoires (`C:\PHP` dans mon cas).

Si vous n'avez pas déjà installé MySQL, consultez à nouveau l'introduction de cet ouvrage dans laquelle la procédure d'installation est détaillée.

Fichiers de configuration

Dans le fichier de configuration d'Apache `httpd.conf` (situé dans `C:\Program Files (x86)\Apache Software Foundation\Apache2.2` dans mon cas), le signe `#` désigne un commentaire. Vous devrez modifier quelques lignes et en ajouter d'autres :

- Ajoutez à la liste existante, la ligne suivante (assurez-vous que ce fichier est bien présent dans le répertoire de PHP) :

```
LoadModule php5_module "C:\php\php5apache2_2.dll"
```

- Modifiez la valeur de `DocumentRoot` (à deux endroits) et indiquez le répertoire qui contiendra vos sources PHP.

```
DocumentRoot "C:/Donnees/dev/PHP-MySQL"
…
<Directory "C:/Donnees/dev/PHP-MySQL">
```

- Ajoutez la valeur `index.php` dans la section `<IfModule dir_module>` :

```
<IfModule dir_module>
DirectoryIndex index.html index.php
</IfModule>
```

- Après la section `<IfModule mime_module> … </IfModule>`, ajoutez les lignes

suivantes :

```
AddType application/x-httpd-php .php .php5
AddType application/x-httpd-php-source .phps
```

Concernant PHP, copiez le fichier `php.ini-development` dans le répertoire de Windows sous le nom `%systemroot%/php.ini` (`C:\Windows` dans mon cas) et décommenter la ligne `extension=php_mysqli.dll`.

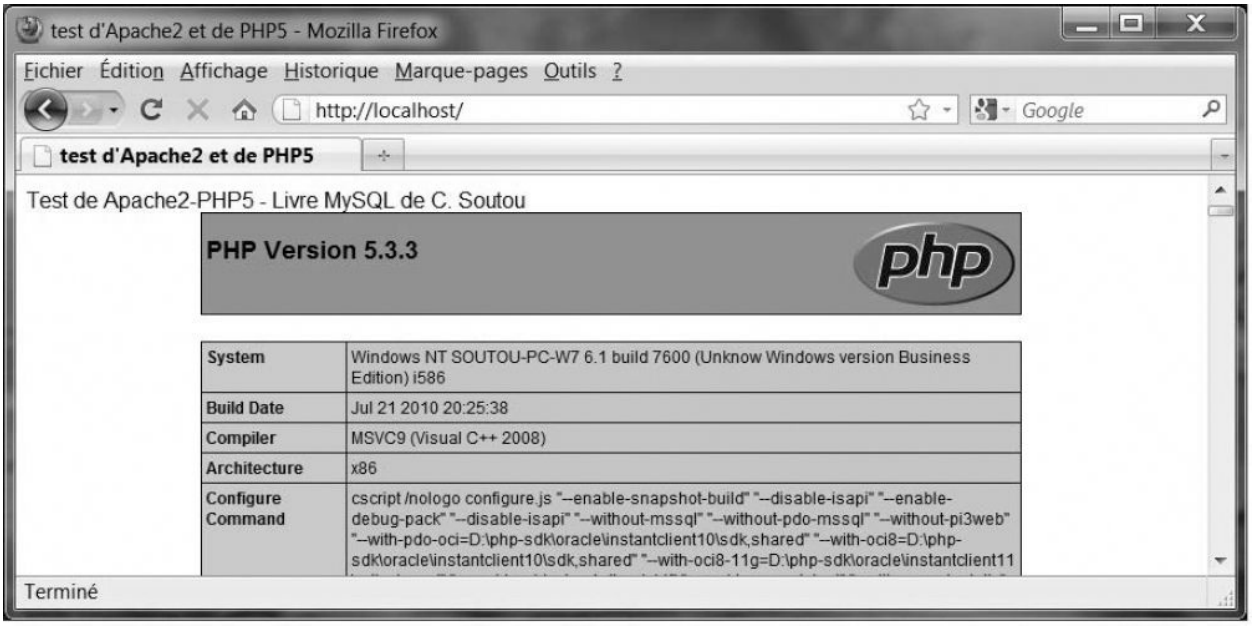
Test d'Apache et de PHP

Testez le programme suivant (`index.php`) après l'avoir déposé dans le répertoire prévu pour contenir les sources PHP (dans mon cas `C:\Donnees\dev\PHP-MySQL`).

```
<html>
<title>test d'Apache2 et de PHP5</title>
<body>
Test de Apache2-PHP5 - Livre MySQL de C. Soutou
<?php
  phpinfo();
?>
</body>
</html>
```

Pour tester le programme, arrêtez Apache, puis relancez-le. Dans le navigateur, saisissez l'URL de votre serveur sur le port concerné (`http://localhost` dans mon cas), qui doit exécuter le programme `index.php`. La configuration actuelle de PHP (résultat de la fonction système PHP `phpinfo`) doit alors s'afficher.

Figure 9-1 *Test d'Apache et de PHP*



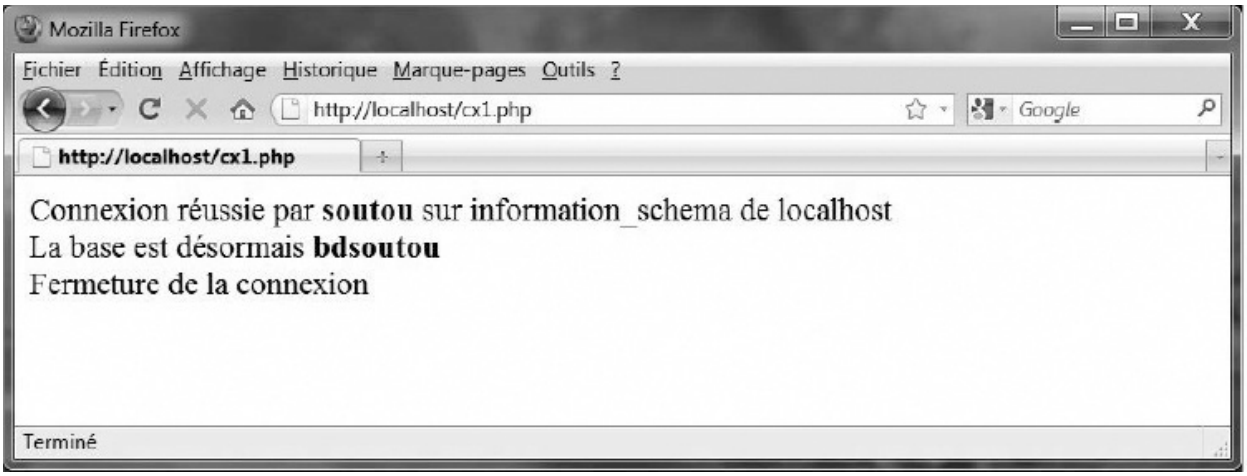
Test d'Apache, de PHP et de MySQL

Le serveur MySQL doit être démarré. Écrivez le programme `cx1.php` suivant et déposez-le dans le répertoire contenant les sources PHP. Renseignez le nom d'utilisateur MySQL, le mot de passe et le nom de la base. Ici, je lance une connexion à la base du dictionnaire des données, puis je sélectionne la base `bdsoutou`.

```
<?php
$service=mysqli_connect('localhost','soutou','iut','information_schema');
if ($service)
 {print "Connexion réussie avec <B>soutou</B> sur information_schema
         de localhost";
  if (($usebdsoutou = mysqli_select_db($service,'bdsoutou')) > 0)
   { print "<BR> La base est désormais <B>bdsoutou</B>"; }
  else
   { print "<BR> Sélection impossible sur <B>bdsoutou</B>"; }
  print "<BR> Fermeture de la connexion";
  mysqli_close($service);
 }
 else
  print "<BR> La connexion est un échec !";
?>
```

Testez ce programme dans le navigateur (`http://localhost/cx1.php` dans mon cas). Vous devez obtenir un résultat analogue à celui de la figure suivante :

Figure 9-2 *Test d'une connexion MySQL*



API de PHP pour MySQL

Ce paragraphe décrit les principales fonctions de l'API `mysqli`. Nous étudierons principalement le mode procédural. La forme objet sera abordée en fin de paragraphe. Afin de tester les exemples, vous trouverez sous `Chapitre9\scriptMySQL` le script pour créer les tables.

Connexion

La fonction `mysqli_connect` retourne un objet de connexion utilisé par la majorité des appels à la base. Les fonctions `mysqli_close`, `mysqli_select_db` et `mysqli_change_user` renvoient `TRUE` en cas de succès, `FALSE` en cas d'erreur. Une connexion se ferme implicitement en fin de programme même si elle n'a pas été clôturée explicitement.

Tableau 9-1 Fonctions de connexion et de déconnexion

Nom de la fonction	Paramètres
`ressource mysqli_`**`connect`**`(string` *`serveur`*`, string` *`utilisateur`*`, string` *`motpasse`* `[,string` *`nomBase`*`])`	Nom du serveur, utilisateur, mot de passe, nom de la base éventuellement. Retourne `FALSE` en cas d'erreur.
`boolean mysqli_`**`close`**`(ressource` *`connexion`*`)`	Ferme la connexion dont l'identifiant passe en paramètre.
`boolean mysqli_`**`select_db`**`(ressource`	Modifie la sélection de la base de la

connexion, string *nomBase*)	connexion dont l'identifiant passe en paramètre.
boolean mysqli_**change_user**(ressource *connexion*, string *utilisateur*, string *motpasse*, string *nomBase*)	Modifie l'utilisateur et la base de la connexion dont l'identifiant passe en paramètre.

Interactions avec la base

La majorité des traitements SQL, lorsqu'ils incluent des paramètres, s'effectuent comme suit : connexion (*connect*), préparation de l'ordre (*parse*), association des paramètres à l'ordre SQL (*bind*), exécution dudit ordre (*execute*), lecture des lignes (pour les SELECT, *fetch*) et libération des ressources (*free* et *close*) après une éventuelle validation de la transaction courante (*commit* ou *rollback*).

Préparation, exécution

La fonction mysqli_prepare prépare l'ordre SQL puis retourne un objet qui peut être utilisé notamment par les fonctions mysqli_stmt_bind_param et mysqli_stmt_execute. La fonction mysqli_prepare retourne FALSE dans le cas d'une erreur, mais ne valide ni sémantiquement ni syntaxiquement l'ordre SQL. Il faudra attendre pour cela son exécution par mysqli_stmt_execute.

La fonction mysqli_stmt_execute exécute un ordre SQL préparé (renvoie TRUE en cas de succès, FALSE sinon). Le mode par défaut est *auto-commit*. Pour la programmation de transactions, désactiver ce mode (avec mysqli_autocommit) puis valider explicitement par mysqli_commit.

La fonction mysqli_stmt_fetch exécute pas à pas une requête préparée (retourne TRUE en cas de succès, FALSE sinon).

Validation

Les fonctions mysqli_commit et mysqli_rollback permettent de gérer des transactions, elles retournent TRUE en cas de succès, FALSE sinon.

Le programme suivant (insert1.php) insère une nouvelle compagnie (en supposant qu'aucune erreur n'est retournée de la part de la base). Nous étudierons plus loin comment passer des paramètres à une instruction

(*prepared statement*) et comment récupérer, au niveau de PHP, les erreurs renvoyées par MySQL.

Tableau 9-2 Fonctions d'analyse et d'exécution

Nom de la fonction	Paramètres
`ressource mysqli_`**`prepare`**`(ressource` *`connexion`*`, string` *`ordreSQL`*`)`	Le premier paramètre désigne l'identifiant de la connexion. Le second contient l'ordre SQL à analyser (SELECT, INSERT, UPDATE, DELETE, CREATE…).
`boolean mysqli_`**`stmt_execute`**`(ressource` *`ordreSQL`*`)`	Le paramètre désigne l'identifiant d'état à exécuter (renvoyé par *prepare*).
`boolean mysqli_`**`stmt_fetch`**`(ressource` *`ordreSQL`*`)`	Affecte aux variables de liaison PHP le résultat d'une requête préparée.

Tableau 9-3 Fonctions de validation et d'annulation

Nom de la fonction	Paramètres
`boolean mysqli_`**`commit`**`(ressource` *`connexion`*`)`	Valide la transaction de la connexion en paramètre.
`boolean mysqli_`**`rollback`**`(ressource` *`connexion`*`)`	Annule la transaction de la connexion en paramètre.

Tableau 9-4 Insertion d'un enregistrement

Code PHP	Commentaires

`<?php`
`$service = mysqli_connect ('localhost','soutou',`
`'iut','bdsoutou');`
`if ($service)`
	Connexion.

`{ mysqli_autocommit($service,FALSE);`	Début de la transaction.
`$insert1 = "INSERT INTO bdsoutou.Compagnie` `VALUES('AL','Air Lib')";`	Création de l'instruction.
`$ordre = mysqli_`**`prepare`**`($service, $insert1);`	Prépare l'insertion.
`if ( ($res = mysqli_`**`stmt_execute`**`($ordre)) > 0)` `{ print "  Ajout opéré";` `mysqli_`**`commit`**`($service); }`	Exécute l'insertion.

```
    mysqli_stmt_free_result($ordre);
```
Validation. Libère les ressources.

```
    mysqli_close($service);
```
Ferme la connexion.

```
  }
  else print "<BR> La connexion est un échec!";
?>
```

Si vous souhaitez connaître le nombre de lignes affectées par l'ordre SQL, utilisez « `mysqli_stmt_affected_rows($ordre)` » (voir la section « Métadonnées »).

Constantes prédéfinies

Les constantes suivantes permettent de positionner des indicateurs jouant le rôle de paramètres système (modes d'exécution) au sein d'instructions SQL. Nous verrons au long de nos exemples l'utilisation de certaines de ces constantes.

Tableau 9-5 Constantes prédéfinies

Constante	Commentaires
`MYSQLI_ASSOC`	Utilisé par `mysqli_fetch_array` afin d'extraire un *associative array* comme résultat.
`MYSQLI_NUM`	Utilisé par `mysqli_fetch_array` afin d'extraire un *enumerated array* comme résultat.
`MYSQLI_BOTH`	Utilisé par `mysqli_fetch_array` afin d'extraire un *array* reconnaissant à la fois le mode associatif et le mode numérique en indices.

Extractions

Les fonctions suivantes permettent d'extraire des données via un curseur que la documentation de PHP appelle *tableau*. On rappelle que MySQL retourne les noms de colonnes toujours en minuscules. Cette remarque intéressera les habitués des tableaux à accès associatifs (exemple : `$tab['prenom']`, `prenom` étant une colonne extraite d'une table).

Tableau 9-6 Fonctions d'extraction

Nom de la fonction	Paramètres
`ressource mysqli_`**`query`**`(ressource` *`ordreSQL`* `[,ressource` *`connexion`*`])`	Exécute la requête sur la base de données en cours. Retourne un identifiant de résultat ou `FALSE` en cas d'erreur.
`array mysqli_`**`fetch_array`**`(ressource` *`idresultat`* `[, int` *`param`*`])`	Retourne un tableau qui contient la ligne du curseur suivante, ou `FALSE` en cas d'erreur ou en fin de curseur. Le tableau est accessible de manière associative ou numérique suivant le paramètre *param* qui peut être une combinaison de : • `MYSQLI_BOTH` (par défaut, identique à `MYSQLI_ASSOC + MYSQLI_NUM`). • `MYSQLI_ASSOC` pour un tableau à accès associatif (comme `mysqli_fetch_assoc`). • `MYSQLI_NUM` pour un tableau à accès numérique (comme `mysqli_fetch_row`).
`array mysqli_`**`fetch_assoc`**`(ressource` *`ordreSQL`*`)`	Retourne la ligne du curseur suivante dans un tableau associatif, ou `FALSE` en cas d'erreur ou en fin de curseur.
`object` **`mysqli_fetch_object`**`(ressource` *`ordreSQL`*`)`	Retourne la ligne du curseur suivante dans un objet PHP ou `FALSE` en cas d'erreur ou en fin de curseur.
`array` **`mysqli_fetch_row`**`(ressource ordreSQL)`	Retourne la ligne du curseur suivante dans un tableau numérique, ou `FALSE` en cas d'erreur ou en fin de curseur.
`boolean mysqli_`**`stmt_free_result`**`(ressource` *`ordreSQL`*`)`	Libère les ressources associées aux curseurs occupés après `mysqli_prepare`. Retourne `TRUE` en cas de succès, `FALSE` dans le cas inverse.

Illustrons à partir d'exemples certaines utilisations de quelques-unes de ces

fonctions.

Le programme suivant (`select1.php`) utilise `mysqli_fetch_array` afin d'extraire les avions de la compagnie de code 'AF'. On suppose ici, et dans les programmes suivants, que la connexion à la base est réalisée et se nomme `$service`. Le curseur obtenu est nommé `ligne`, il prend en compte les valeurs nulles éventuelles.

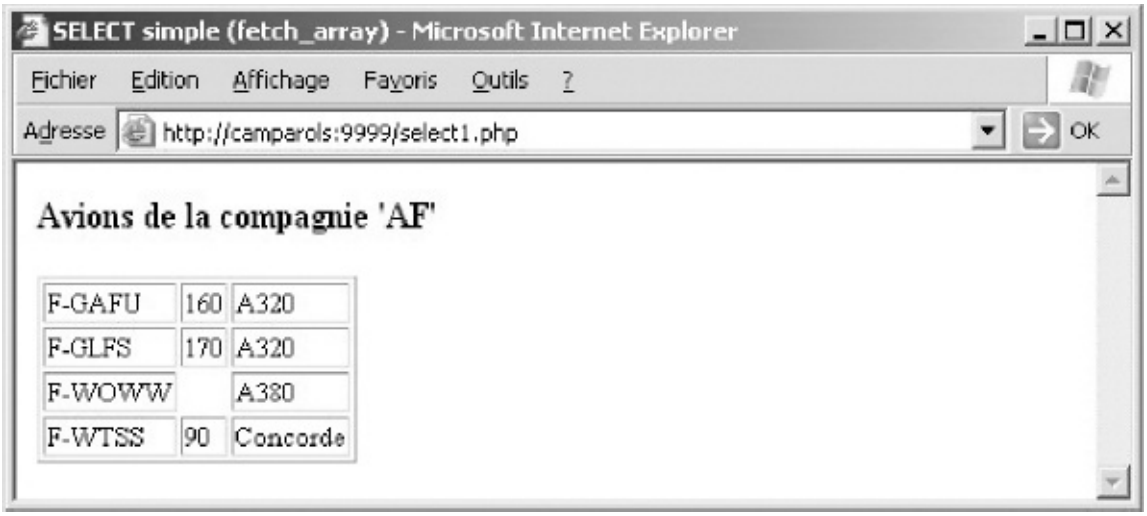
Tableau 9-7 Extraction à l'aide de `mysqli_fetch_array`

Code PHP	Commentaires
`$requete = "SELECT immat,capacite,typeAvion FROM Avion WHERE compa = 'AF'";`	Création de la requête.
`if ($resultat=mysqli_query($service, $requete))`	
`{$ncols = mysqli_num_fields($resultat);`	Chargement du curseur.
`print "<H3>Avions de la compagnie 'AF'</H3>";` `print "<TABLE BORDER=1> ";`	Obtention du nombre de colonnes.
`while ($ligne = mysqli_fetch_array($resultat))`	Parcours des colonnes.
`{print "<TR> ";` `for ( $i=0;$i < $ncols; $i++)` `{print "<TD> $ligne[$i] </TD>" ;}`	
`print "</TR> "; }` `print "</TABLE> ";`	Affichage des colonnes.
`mysqli_free_result($resultat);`	Libération des ressources.
`}` `else` `print " La requête est un échec!";` `mysqli_close($service);`	
	Fermeture de la connexion.

- La fonction `mysqli_num_fields` renvoie le nombre de colonnes de la requête et sa signature est détaillée à la section « Métadonnées ».
- La fonction `mysqli_num_rows($resultat)` aurait retourné 4 (nombre de lignes extraites).
- La fonction `mysqli_free_result` joue le même rôle que `mysqli_stmt_free_result`, mais s'applique aux instructions `SELECT`.

Vous devez obtenir un résultat analogue (en supposant que la compagnie `'AF'` dispose de quatre avions dont un est affecté d'une capacité nulle). Le test est réalisé sur une autre machine (de nom `camparols` avec Apache configuré sur le port 9999).

***Figure 9-3** Exemple avec* `mysqli_fetch_array`



Le programme suivant (`select2.php`) décrit l'utilisation de la fonction `mysqli_fetch_assoc` pour extraire tous les Airbus. Le tableau associatif obtenu est nommé `row`. L'accès à chaque cellule de ce tableau est réalisé à l'aide du nom des colonnes.

Tableau 9-8 Extraction à l'aide de `mysqli_fetch_assoc`

Code PHP	Commentaires
`$requete = "SELECT immat,typeAvion,capacite FROM Avion WHERE typeAvion LIKE 'A%'";`	Création de la requête.
`if ($resultat=mysqli_query($service, $requete))`	Exécution de la requête.

`{print "<H3>Liste des Airbus</H3>";`
`print "<table border=1>\n";`
`print "<tr><td>Imatriculation</td>`
`<td>Type</td><td>capacite</td>";`
`$i=0;`
`while ($row = mysqli_fetch_assoc($resultat))`	

`{print "<tr><td>".$row["immat"].`
`"</td><td>".$row["typeAvion"].`
`"</td><td>".$row["capacite"]."</td></tr>"; }`
	Parcours du curseur.
	Affichage des colonnes.

`print "</table>\n";` `mysqli_free_result($resultat);`	

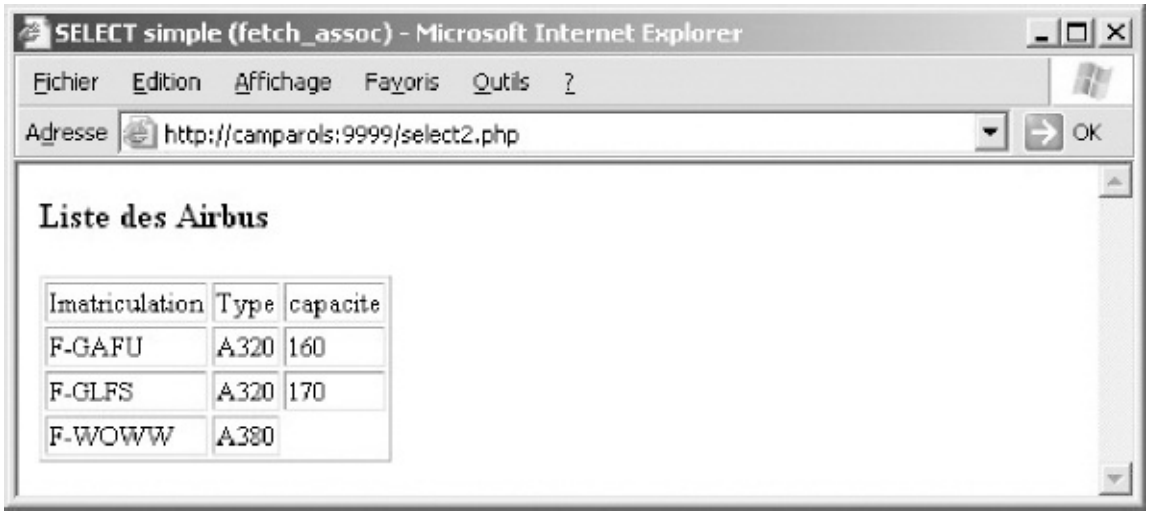
```
 }
  else { print "<BR>La requête est un échec!"; }
mysqli_close($service);
```

Libération des ressources.

Fermeture de la connexion.

Le résultat est le suivant (en supposant que la base ne stocke que trois avions de type Airbus) :

Figure 9-4 *Exemple avec* `mysqli_fetch_assoc`



Instructions paramétrées

Les fonctions `mysqli_stmt_bind_param` et `mysqli_stmt_bind_result` permettent d'associer à des colonnes MySQL des variables PHP et inversement. Ces fonctions retournent `TRUE` en cas de succès, `FALSE` sinon.

Tableau 9-9 Fonctions de passage de paramètres

Nom de la fonction	Paramètres
`boolean mysqli_stmt_bind_result(ressource ordreSQL, mixed &variable1, [,mixed &variable2…])`	Le premier paramètre est l'identifiant d'état obtenu après *prepare*. Les paramètres suivants listent les variables PHP de réception.
	Le premier paramètre est l'identifiant de résultat obtenu après

`boolean mysqli_stmt_bind_param(ressource ordreSQL, string types, mixed &variable1 [,mixed &variable2…])`	*prepare*. Le deuxième paramètre décrit les types des variables à substituer aux colonnes. Les paramètres suivants listent les variables PHP en entrée.
`string types`	Description des types des variables (`i` pour numérique, `d` pour flottant, `s` pour chaîne de caractère et `b` pour *BLOB*). Concaténer autant de ces caractères qu'il existe de variables.

Extractions

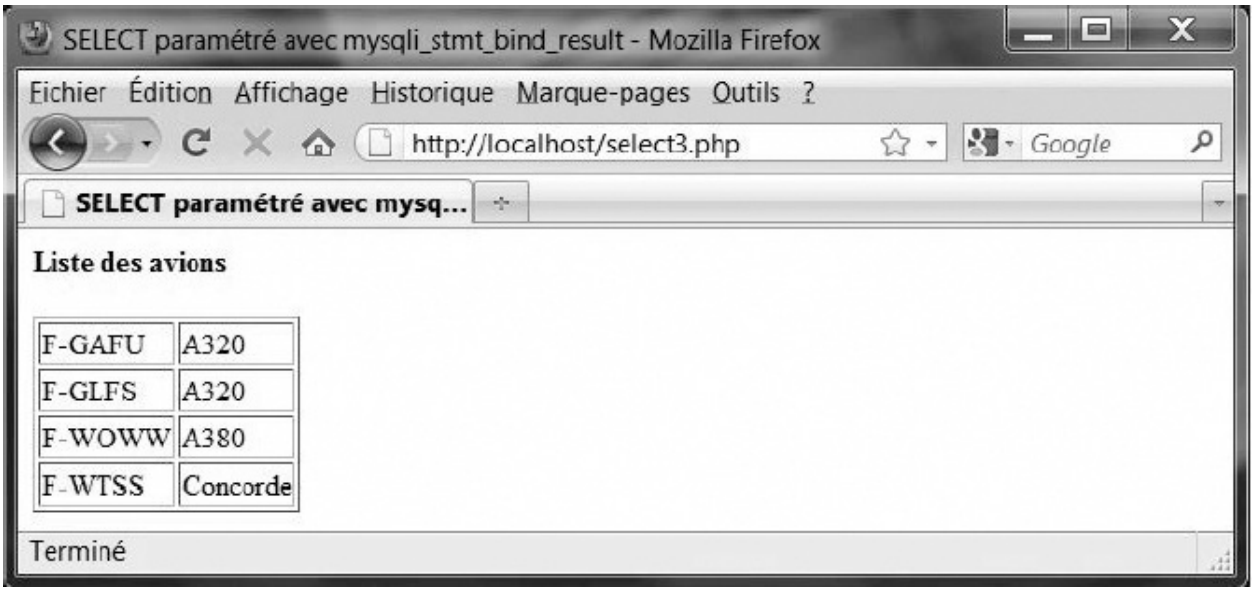
Le programme suivant (`select3.php`) utilise la fonction `mysqli_stmt_bind_result` afin d'extraire l'immatriculation et le type de tous les avions (au travers de variables PHP qui sont définies après exécution de la requête). Notez l'utilisation des fonctions `mysqli_stmt_fetch` pour parcourir le résultat de la requête ligne après ligne, et `mysqli_stmt_close` pour fermer la requête préparée.

Tableau 9-10 Extraction préparée avec la fonction `mysqli_stmt_bind_result`

Code PHP	Commentaires
`$requete = "SELECT immat,typeAvion FROM bdsoutou.Avion";` `$ordre = mysqli_prepare($service,$requete);` `if ( ($res = mysqli_stmt_execute($ordre)) > 0)` `{if ( ($resbind =` `mysqli_stmt_bind_result($ordre,$im,$ty)) > 0)` `{print "<H4>Liste des avions</H4>";` `print "<TABLE BORDER=1> ";` `while (mysqli_stmt_fetch($ordre))` `{print "<TR> <TD> $im</TD>" ;` `print " <TD> $ty</TD> </TR> "; }` `print "</TABLE> ";` `} else print " La liaison est un échec!";` `} else print " La requete est un échec!";` `mysqli_stmt_close($ordre);` `mysqli_close($service);`	Définition et exécution de la requête préparée. Affectation des variables PHP. Parcours de la requête. Fermeture de la requête et de la connexion.

Vous devez obtenir un résultat analogue à celui présenté à la figure suivante :

***Figure 9-5** Exemple avec `mysqli_stmt_bind_result`*



Manipulations

Le programme suivant (`insert2.php`) utilise la fonction `mysqli_stmt_bind_param` en faisant passer deux paramètres (variables PHP) lors de l'insertion d'une nouvelle compagnie. On retrouve la notion de *placeholders* (symbole « ? » désignant une valeur d'une colonne d'une table ou d'une vue) étudiée au chapitre 7. Notez le second paramètre de la fonction `mysqli_stmt_bind_param` (« `ss` » désigne deux types chaînes de caractères).

Tableau 9-11 Insertion paramétrée avec la fonction `mysqli_stmt_bind_param`

Code PHP	Commentaires
`mysqli_autocommit($service,FALSE);`	
`$codeComp = "CAST";`	Affectation des variables PHP.
`$nomComp  = "Castanet Air";`	Définition de l'ordre paramétré.
`$insert2  = "INSERT INTO Compagnie VALUES(?,?)";` `$ordre    = mysqli_prepare($service, $insert2);`	
`if ((mysqli_stmt_bind_param($ordre, 'ss', $codeComp,` `$nomComp)) > 0)`	Association avec les variables PHP.
`{if (($res = mysqli_stmt_execute($ordre)) > 0)` `{print " Compagnie $nomComp insérée";` `mysqli_commit($service);`	Exécution de l'ordre.
`mysqli_stmt_free_result($ordre); }`	Validation.

`  else print " L'insertion est un échec!";` `}` `else print " Problème au bind!";` `mysqli_close($service);`	Libération des ressources. Fermeture de la connexion.

Pour toute extraction par `SELECT`, modification par `UPDATE` ou suppression par `DELETE`, le principe à adopter est le même.

Gestion des séquences

La fonction `mysqli_insert_id($connexion)` retourne la valeur courante de la séquence après avoir inséré un enregistrement dans une table contenant une colonne `AUTO_INCREMENT`.

Le programme suivant (`insert3.php`) insère un affrètement (table décrite aux chapitres 2 et 8) à la date du jour pour la compagnie de code 'CAST' et l'avion immatriculé 'F-GRTC'. On récupère la dernière valeur de la séquence après l'insertion.

Tableau 9-12 Insertion paramétrée avec la fonction `mysqli_insert_id`

Code PHP	Commentaires

`mysqli_autocommit($service,FALSE);`
`$codeComp = "CAST";`
`$immatric = "F-GRTC";`
`$n        = 162;`
`$insert3 = "INSERT INTO bdsoutou.Affreter`
`   (comp,immat,dateAff,nbPax)`
`VALUES(?,?,SYSDATE(),?)";`
`$ordre   = mysqli_prepare($service, $insert3);`
	Affectation des variables PHP.

`if ((mysqli_stmt_bind_param($ordre, 'ssi',` `$codeComp,)`	Définition de l'ordre paramétré.
`      $immatric, $n)) > 0)` ` {if (($res = mysqli_stmt_execute($ordre)) > 0)` `   {print " Affretement insérée, sequence : "` `         .mysqli_insert_id($service);` `    mysqli_commit($service);` `    mysqli_stmt_free_result($ordre); }`	Association avec les variables PHP. Exécution de l'ordre. Récupération de la séquence. Validation.
`  else print " L'insertion est un échec!";` `}` `else  print " Problème au bind!";` `  mysqli_close($service);`	Libération des ressources. Fermeture de la connexion.

Traitement des erreurs

Les fonctions `mysqli_errno` et `mysqli_error` permettent de gérer les erreurs retournées par MySQL au niveau de la connexion. Les fonctions `mysqli_stmt_errno` et `mysqli_stmt_error` sont analogues au niveau d'une instruction préparée.

Tableau 9-13 Fonctions pour la gestion des erreurs MySQL

Nom de la fonction	Paramètres
`int mysqli_`**`errno`**`(ressource` *`connexion`*`)`	Retourne le code d'erreur du dernier appel à la base, sur la connexion désignée dans le paramètre (0 si aucune erreur n'est survenue lors du dernier échange).
`string mysqli_`**`error`**`(ressource` *`connexion`*`)`	Retourne le libellé de l'erreur lors du dernier échange (chaîne vide si aucune erreur).
`int mysqli_`**`stmt_errno`** `(ressource` *`ordreSQL`*`)`	*Idem* `mysqli_errno` à partir de l'identifiant d'état désigné dans le paramètre.
`string mysqli_`**`stmt_error`** `(ressource` *`ordreSQL`*`)`	*Idem* `mysqli_error` à partir de l'identifiant d'état désigné dans le paramètre.
`string mysqli_connect_error()`	Retourne le message d'erreur d'une mauvaise connexion.
`int mysqli_connect_errno()`	Retourne le code d'erreur d'une mauvaise connexion.

Le programme suivant (`erreur1.php`) utilise plusieurs de ces fonctions. Dans cet exemple, la suppression ne se déroule pas correctement du fait de l'existence d'enregistrements « fils » dans la table `Avion`. Il est donc possible de dérouter le programme en fonction du code d'erreur MySQL renvoyé (comme pour les exceptions des procédures cataloguées).

Tableau 9-14 Gestion d'erreurs

Code PHP	Commentaires

`$service = mysqli_connect('localhost', 'soutou','iut', 'bdsoutou');`
`if ($service)`
`{$delete1 = "DELETE FROM bdsoutou.Compagnie";`
	Connexion.

```
   $ordre   = mysqli_prepare($service, $delete1);
   print $delete1;

   if ( ($res = mysqli_stmt_execute($ordre)) > 0)
    { print "<BR>Suppression de Compagnie!"; }
   else

    {print "<BR><B>Message : </B>".
mysqli_stmt_error($ordre);

     print "<BR><B>Code : </B>". mysqli_stmt_errno($ordre); }
   mysqli_stmt_free_result($ordre);

   mysqli_close($service);
  }
 else
  {print "L'utilisateur n'a pu se connecter <BR>";
   print "<B>Message : </B>".mysqli_connect_error(); }
```

Préparation de l'ordre.

Exécution.

Affichage de l'erreur.

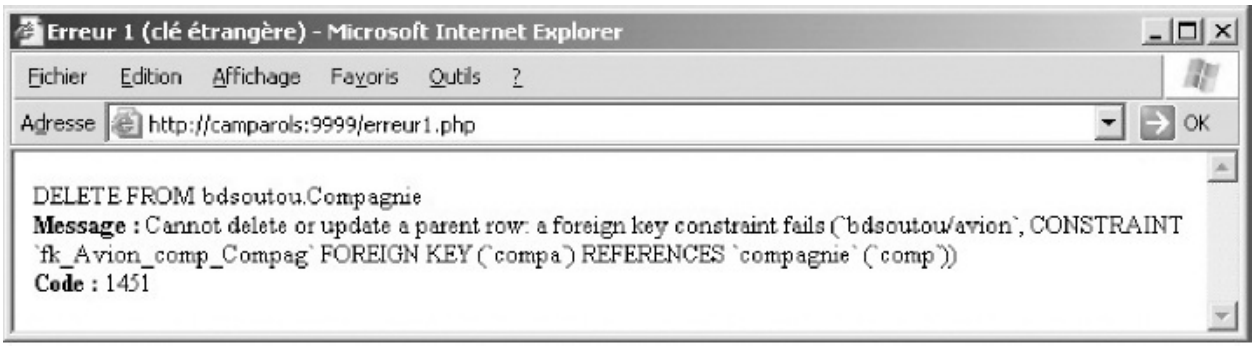
Libération des ressources.

Fermeture de la connexion.

Erreur de connexion.

Le résultat est le suivant :

Figure 9-6 *Exception SQL levée à l'aide de PHP*



Procédures cataloguées

Comme dans tout autre langage hôte, PHP permet d'invoquer des procédures cataloguées côté serveur. Supposons que nous disposions de la procédure `augmenteCap` qui augmente la capacité (premier paramètre) des avions d'une compagnie donnée (deuxième paramètre).

```
CREATE PROCEDURE bdsoutou.augmenteCap(IN nbre TINYINT, IN compag CHAR(4))
BEGIN
  UPDATE Avion SET capacite = capacite + nbre WHERE compa = compag;
END;
```

Paramètre en entrée

Le code suivant (procedureCat.php) appelle cette procédure afin d'augmenter de 50 la capacité des avions de la compagnie de code 'AF' en utilisant la fonction mysqli_multi_query. Notez l'utilisation des simples guillemets pour les paramètres en chaînes de caractères.

Pensez à donner l'autorisation à l'utilisateur appelant (ici soutou) d'exécuter la procédure (je l'ai écrite sous root : GRANT EXECUTE ON PROCEDURE bdsoutou.augmenteCap TO 'soutou'@'localhost'$).

Tableau 9-15 Appel d'une procédure cataloguée (paramètres d'entrée)

Code PHP	Commentaires
`$service = mysqli_connect('localhost', 'soutou','iut', 'bdsoutou');` `if (($service)` `{$nb = 50;` `$comp = 'AF';` `if ($result = mysqli_multi_query($service,` `"call bdsoutou.augmenteCap($nb,'$comp')") > 0)` `{ print " Procédure réalisée correctement."; }` `else` `{ print " La procédure est un échec! ". mysqli_error($service); }` `mysqli_close($service);` `}` `else print " La connexion est un échec!";`	Connexion. Initialisation des variables PHP d'appel. Appel de la procédure. Fermeture de la connexion.

Paramètre en sortie

Afin de travailler avec des paramètres en sortie, il est nécessaire d'utiliser des variables de session. Une fois ces variables de session initialisées au retour de l'appel du sous-programme, il faudra extraire chaque valeur à l'aide d'un SELECT dans une requête simple.

Le code suivant (procedureCat2.php) décrit l'appel de la procédure leNomCompagnieEst (décrite au chapitre 7, section « Procédures cataloguées ») ayant deux paramètres. Le premier (en entrée) indique l'avion de la compagnie recherché, le second (en sortie) contient le nom de la compagnie.

Notez l'utilisation de :

- la fonction `mysqli_multi_query` pour appeler la procédure cataloguée ;
- la variable de session `@v_retour` pour récupérer le paramètre en sortie ;
- la fonction `mysqli_query` pour extraire la valeur de la variable de session (contenue à l'indice 0 du tableau résultat, car il n'y a ici qu'une valeur retournée dans le `SELECT`).

Tableau 9-16 Appel d'une procédure cataloguée (paramètre de sortie)

Code PHP	Commentaires
`$service = mysqli_connect('localhost', 'soutou', 'iut', 'bdsoutou');` `if ($service)`	Connexion.
`{$immat = 'F-GAFU';` `  if ($result = mysqli_multi_query`	Initialisation de la variable PHP d'appel.

`$service,"call bdsoutou.leNomCompagnieEst ('$immat',@v_retour)")  > 0)`
`{if ($result2 = mysqli_query($service,`
`"SELECT @v_retour"))`
	Appel de la procédure.

`{$ligne = mysqli_fetch_array($result2, MYSQLI_NUM);` `if ($ligne[0] == null)` `print " Désolé, l'avion $immat n'a pas de compagnie!";` `else` `print " La compagnie de l'avion $immat est :". $ligne[0];`	Extraction de la variable de session.
`mysqli_free_result($result2);` `}` `else {print " Problème au retour du paramètre ".` `mysqli_error($service); }`	Affichage du résultat.
`}` `else {print " La procédure est un échec! ".` `mysqli_error($service).$result; }`	Gestion des erreurs.
`mysqli_close($service);` `}` `else print " La connexion est un échec!";`	Fermeture de la connexion.

Métadonnées

Plusieurs fonctions permettent d'extraire des informations en provenance du dictionnaire des données (*meta data*) à partir d'une instruction SQL. Par exemple, `mysqli_stmt_result_metadata` retourne un identifiant de résultat permettant d'extraire des métadonnées à partir d'une requête préparée. La fonction `mysqli_fetch_field` retourne un objet qui contient les métadonnées des colonnes concernées par une requête.

Tableau 9-17 Fonctions pour les métadonnées

Nom de la fonction	Paramètres
`ressource mysqli_stmt_result_metadata (ressource ordreSQL)`	Le paramètre est l'identifiant d'état obtenu après *prepare*.
`object mysqli_fetch_field(ressource ordreSQL)`	Le paramètre est l'identifiant de résultat obtenu après `mysqli_query`, ou `FALSE` si aucune information n'est renvoyée pour l'ordre SQL concerné.
`int mysqli_num_fields(ressource ordreSQL)`	Retourne le nombre de colonnes concernées par l'ordre SQL. Le paramètre est l'identifiant de résultat obtenu après `mysqli_query`.

Citons d'autres fonctions comme `mysqli_fetch_field_direct` et `mysqli_fetch_fields` qui sont similaires à `mysqli_fetch_field`. Une fois l'objet retourné par cette fonction, il faut extraire certains de ces champs à la demande.

Tableau 9-18 Champs de l'objet retourné par `mysqli_fetch_field`

Nom du champ	Signification
`name`	Nom de la colonne.
`table`	Nom de la table à laquelle la colonne appartient (si elle n'a pas été calculée).
`def`	Valeur par défaut de la colonne.
`max_length`	Taille maximale de la colonne pour le jeu de résultats retourné par la requête.
`flags`	Entier représentant le *bit-flags* pour la colonne (codage des contraintes).
`type`	Code du type de données de la colonne.
`decimals`	Nombre de décimales de la colonne.

Tableau 9-19 Code du type de données

Type MySQL	Code

	équivalent
DECIMAL	0
TINYINT	1
SMALLINT	2
INT	3
FLOAT	4
DOUBLE	5
TIMESTAMP	7
BIGINT	8
MEDIUMINT	9
DATE	10
TIME	11
DATETIME	12
YEAR	13
TEXT, TINYBLOB, TINYTEXT, BLOB, MEDIUMBLOB, MEDIUMTEXT, LONGBLOB, LONGTEXT	252
VARBINARY, VARCHAR	253
CHAR, BINARY, ENUM, SET	254

Fonction mysqli_fetch_field

Le programme suivant (`meta1.php`) utilise la fonction `mysqli_fetch_field` afin d'extraire la structure complète (en termes de colonnes) d'une table.

Tableau 9-20 Extraction de la structure d'une table

Code PHP	Commentaires

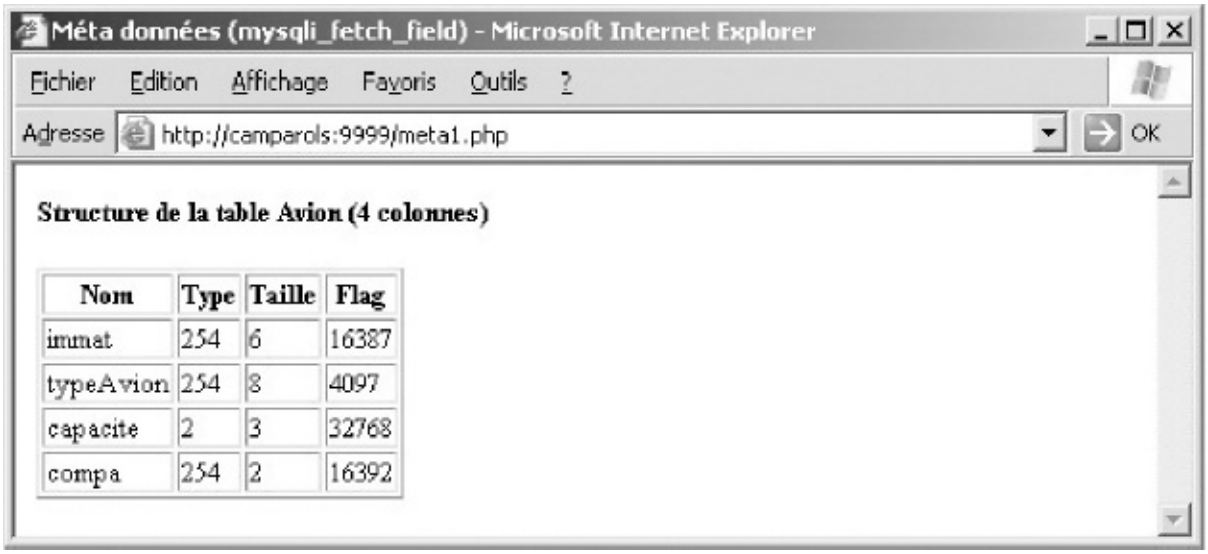
`$res = mysqli_query($service, "SELECT * FROM Avion");`
`if ($res)`
`{$ncols = mysqli_num_fields($res);`
`print "<H4>Structure de la table Avion ($ncols colonnes)</H4>";`
`print "<table border=1>";`
`print "<tr><th>Nom</th><th>Type</th><th>Taille</th><th>Flag</th></tr>";`
 	
`while ($obj=mysqli_fetch_field($res))`
`print "<tr><td>$obj->name</td><td>$obj->type</td><td>$obj->max_length</td><td>$obj->flags</td></tr>";`
`print "</table>\n";`
	Requête.
	
	Extraction du nombre de colonnes.
	
	Affichage du nom, code du type, taille maximale et contraintes de

```
 }
 else print "<BR>La requete est un échec!";
mysqli_close($service);
```

chaque colonne extraite.

Le résultat est le suivant :

Figure 9-7 *Extraction de la structure d'une table*



Quelques explications :

- La taille vaut 2 pour la colonne `compa`, car seule la compagnie de code 'AF' est représentée dans mon jeu d'essai.

```
(root@localhost) [bdsoutou] mysql> select * from avion;
+--------+-----------+----------+-------+
| immat  | typeAvion | capacite | compa |
+--------+-----------+----------+-------+
| F-GAFU | A320      |      160 | AF    |
| F-GLFS | A320      |      170 | AF    |
| F-WOWW | A380      |     NULL | AF    |
| F-WTSS | Concorde  |       90 | AF    |
+--------+-----------+----------+-------+
```

- Le *flag* vaut 16 387 pour la colonne `immat`, car cette colonne est une clé primaire (ajouter 1) et non nulle (ajouter 2), et elle fait partie d'une clé (ici primaire, ajouter 16 384). Ci-après, un extrait du fichier `mysql_conf.h`. Le *flag* vaut 16 392 pour la colonne `compa`, car elle fait partie d'une clé (ici étrangère, ajouter 16 384), et c'est une clé étrangère (ajouter 8), etc.

```
#define NOT_NULL_FLAG    1            /* Field can't be NULL */
#define PRI_KEY_FLAG     2            /* Field is part of a primary key */
#define UNIQUE_KEY_FLAG 4             /* Field is part of a unique key */
#define MULTIPLE_KEY_FLAG 8           /* Field is part of a key */
```

```
#define BLOB_FLAG       16          /* Field is a blob */
#define UNSIGNED_FLAG   32          /* Field is unsigned */
#define ZEROFILL_FLAG   64          /* Field is zerofill */
#define BINARY_FLAG     128         /* Field is binary   */
#define ENUM_FLAG       256         /* field is an enum */
#define AUTO_INCREMENT_FLAG 512     /* field is a autoincrement field */
#define TIMESTAMP_FLAG  1024        /* Field is a timestamp */
#define SET_FLAG        2048        /* field is a set */
#define NO_DEFAULT_VALUE_FLAG 4096  /* Field doesn't have default value */
#define PART_KEY_FLAG   16384       /* Intern; Part of some key */
#define NUM_FLAG        32768       /* Field is num (for clients) */
```

Fonction mysqli_stmt_result_metadata

Peu de changements par rapport au programme précédent. La fonction `mysqli_stmt_result_metadata` suppose qu'on travaille avec un état préparé. Elle sert à affecter un identifiant de résultat qui passe en paramètre de `mysqli_fetch_field`, comme vu précédemment.

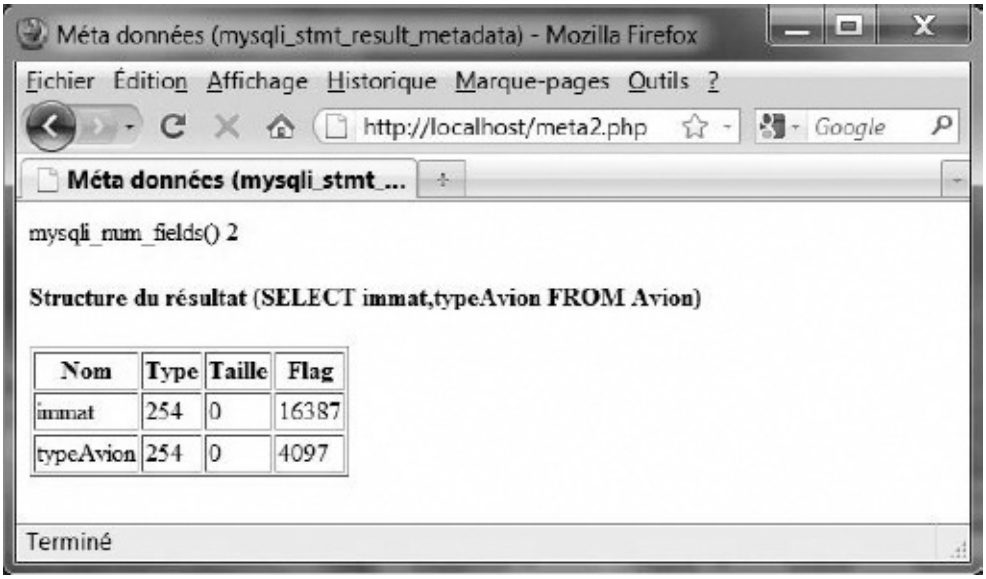
Le programme suivant (`meta2.php`) illustre cette fonction afin d'extraire la structure incomplète en termes de colonnes d'une table.

Tableau 9-21 Extraction d'une partie de la structure d'une table

Code PHP	Commentaires
`$requete = "SELECT immat,typeAvion FROM Avion";`	Requête.
`$ordre = mysqli_prepare($service,$requete);` `$res = mysqli_stmt_result_metadata($ordre);` `if ($res)` `{ $ncols = mysqli_num_fields($res);` `print "mysqli_num_fields() ".$ncols ;` `print " <H4>Structure du résultat ($requete)</H4>";` `print "<table border=1>";` `print "<tr>"; print "<th>Nom</th>";` `print "<th>Type</th>"; print "<th>Taille</th>";` `print "<th>Flag</th>"; print "</tr>";`	Extraction du nombre de colonnes.
`while ( $obj = mysqli_fetch_field($res) )` `{print "<tr>"; print "<td>$obj->name</td>";` `print "<td>$obj->type</td>";` `print "<td>$obj->max_length</td>";` `print "<td>$obj->flags</td>"; print "</tr>"; }` `print "</table>\n";` `}` `else print " La requête est un échec !";` `mysqli_close($service);`	Affichage du nom, code du type, taille maximale et contraintes de chaque colonne extraite.

Le résultat est le suivant :

***Figure 9-8** Exemple avec* `mysqli_stmt_result_metadata`



Style d'écriture objet

La classe `MySQLi` (http://www.php.net/manual/fr/class.mysqli.php) regroupe toutes les fonctions disponibles avec l'API `mysqli` sous la forme de méthodes et d'attributs. Une fois créée une instance de cette classe à l'aide des informations de connexion à une base `$objet=new MySQLi('serveur','utilisateur','motdepasse','nomBase')`, il suffit d'invoquer chaque méthode sous la forme `$objet->methode(parametres)`. À titre d'exemple, le programme suivant réalise une connexion, un changement de base et une déconnexion ou l'affichage de l'erreur le cas échéant.

```
<?php
$objet = new MySQLi('localhost','soutou',iut,'information_schem
if ($objet->connect_error)
{ print "<BR> La connexion est un échec !";
  print "<BR> Erreur : . $objet->connect_errno";
  print "<BR> Erreur : . $objet->connect_error"; }
else
{print "Connexion réussie";
 if ( $objet->select_db('bdsoutou'))
    {  print "<BR> La base est désormais <B>bdsoutou</B>"; }
 else
    {  print "<BR> Sélection impossible sur <B>bdsoutou</B>"; }
       print "<BR> Fermeture de la connexion";
       $objet->close(); }
?>
```

D'autres classes sont proposées, citons `MySQLi_STMT` pour gérer les requêtes

préparées et `MySQLi_Result` pour manipuler les résultats obtenus d'une requête. Vous trouverez à l'adresse suivante http://www.php.net/manual/fr/mysqli.summary.php les équivalences entre les procédures de l'API `mysqli` que nous avons étudiées et les signatures des méthodes objet.

Exercices

L'objectif de ces exercices est de compléter des progammes PHP pour extraire et mettre à jour des données de certaines des tables du schéma *Parc informatique*.

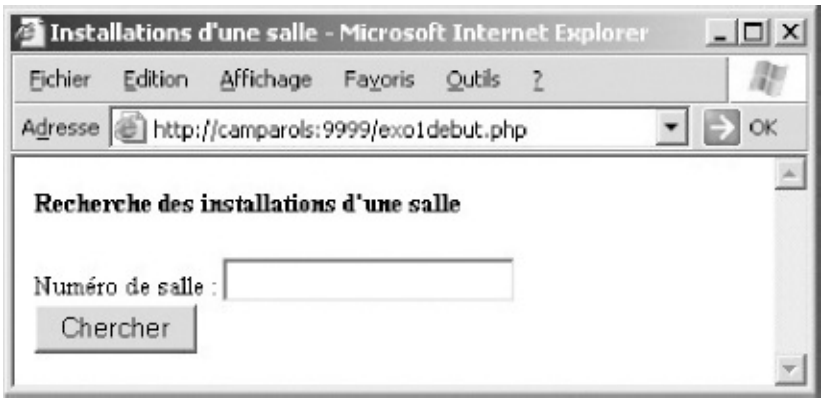
Exercice 9.1

Extraction préparée

Écrire le programme `exo1suite.php` qui devra retrouver le nom du logiciel, le numéro de poste, la date d'installation et la date d'achat du logiciel pour toutes les installations d'une salle donnée. Ce programme sera appelé à partir du programme `exo1debut.php` composant un formulaire de saisie :

```
<html> <head> <title>Installations d'une salle</title> </head>
<body>
<form  action="./exo1suite.php" method="POST"><p>
 <H4>Recherche des installations d'une salle</H4><P>
  Numéro de salle : <input type="text" name="ns" maxlength="7"/><BR>
  <input type="submit" value="Chercher"/>
</form>
</body> </html>
```

Figure 9-9 *Formulaire de saisie*



Vous utiliserez :

• `$_POST['ns']` pour récupérer la valeur du numéro de salle saisi dans le formulaire.

• `mysqli_prepare`, `mysqli_stmt_execute` et `mysqli_stmt_bind_result` pour préparer, exécuter et lier des variables PHP en sortie.

• Un affichage simple de type « Aucune installation dans la salle, si la salle ne contient aucun poste sur lequel un logiciel est installé » (exemple pour 's12')

Votre programme doit produire un résultat analogue à l'écran suivant :

Figure 9-10 *Extraction préparée*

Exercice 9.2

Appel d'un sous-programme

Utiliser de nouveau la procédure cataloguée `supprimeSalle(IN ns VARCHAR(7),OUT res TINYINT)` décrite à l'exercice du chapitre 8. Cette procédure supprime une salle dont le numéro est passé en premier paramètre et retourne en second paramètre :

• 0 si la suppression s'est déroulée correctement ;

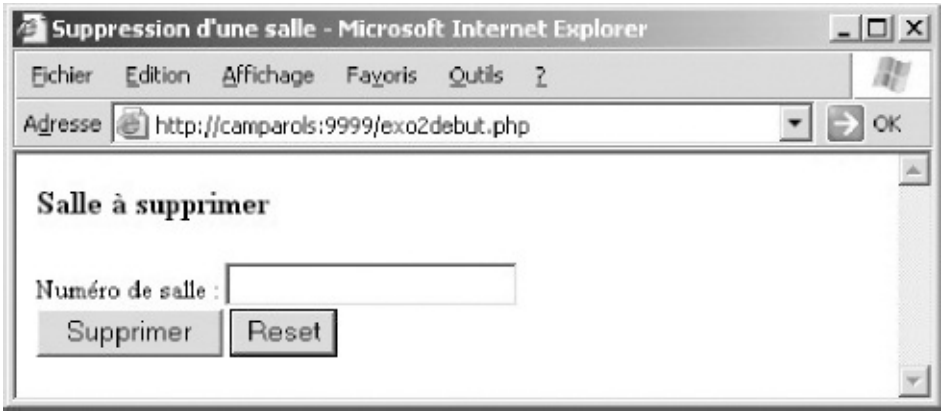
• – 1 si le code de la salle est inconnu ;

• – 2 si la suppression est impossible (contraintes référentielles).

Écrire le programme `exo2suite.php` qui, à partir d'une saisie du numéro de salle (programme `exo2debut.php` similaire à `exo1debut.php`), appelle le sous-programme `supprimeSalle`.

```
<html> <head> <title>Suppression d'une salle</title> </head>
<body>
<form action ="./exo2suite.php" method="POST"><p>
  <H3>Salle à supprimer</H3><P>
  Numéro de salle : <input type="text" name="ns" maxlength="7"/><BR>
  <input type="submit" value="Supprimer"/>
  <input type="reset" value="Reset"/>
</form>
</body> </html>
```

***Figure 9-11** Formulaire de saisie*



Tracez les différents cas d'erreurs (numéro de salle référencé par un poste de travail puis numéro de salle inexistant). Pensez à donner à l'utilisateur le privilège en exécution sur cette procédure.

Exercice 9.3

Insertion préparée

Écrire le programme PHP `exo3suite.php` qui, à partir d'une saisie des paramètres nécessaires pour enregistrer une nouvelle installation à la date du jour d'un logiciel sur un poste de travail, réalise l'insertion dans la table `Installer`. Cette saisie sera réalisée dans le programme `exo3debut.php` ci-après :

```
<html> <head> <title>Nouvelle installations de logiciel sur un
poste</title> </head>
<body>
< form action="./exo3suite.php"  method="POST"><p>
 <?php
  $format='j-n-Y';
  $datej = date($format);
  print "<H4>Installation d'un logiciel à la date du $datej</H4><P>";
```

```
 ?>
 Numéro de poste : <input type="text" name="np" maxlength="7"/>
 Code du logiciel : <input type="text" name="nl" maxlength="5"/><BR>
 <input type="submit" value="Enregistrer"/>
 <input type="reset"  value="Reset"/>
</form>
</body> </html>
```

Figure 9-12 *Saisie du formulaire*

Pour tester une éventuelle erreur de compatibilité entre le type du poste et celui du logiciel (voir le déclencheur de l'exercice en fin de chapitre 7), vous afficherez le message d'erreur (avec `mysqli_stmt_error`) et le code d'erreur si l'insertion se passe mal. Tester une insertion correcte ('p10', 'log1') et une insertion incorrecte du fait des types différents ('p8', 'log7').

Figure 9-13 *Insertion correcte*

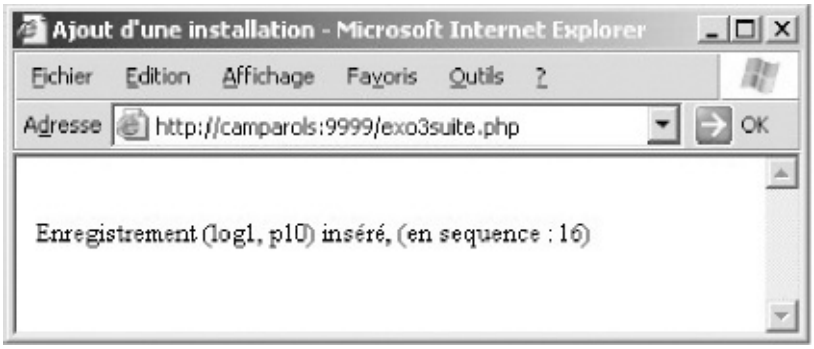
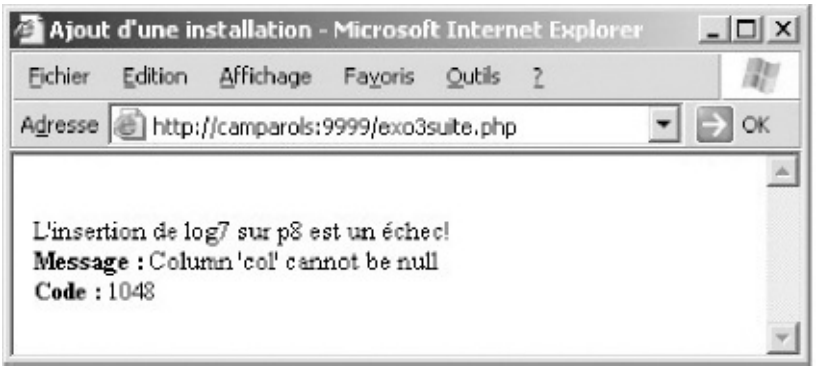


Figure 9-14 *Erreur après insertion*


Ajout d'une installation - Microsoft Internet Explorer
Fichier Edition Affichage Favoris Outils ?
Adresse http://camparols:9999/exo3suite.php
OK
L'insertion de log7 sur p8 est un échec!
Message : Column 'col' cannot be null
Code : 1048

Chapitre 10

Optimisations

Ce chapitre est consacré à l'optimisation du point de vue du développeur. Différents aspects sont étudiés tels que le fonctionnement de l'optimiseur, l'utilisation de statistiques et les plans d'exécution. Des mécanismes utiles sont également présentés : contraintes, index, tables temporaires et partitionnement. Les aspects plus système de l'optimisation, comme la gestion de la mémoire, des transactions, caches, fichiers de trace (*logs*), paramètres d'initialisation, etc., ne seront pas abordés.

Cadre général

Considérons tout d'abord le postulat suivant : « Si une application ralentit les processus métiers, elle doit être optimisée ». La problématique des performances concerne n'importe quelle application et il est normal d'y consacrer du temps, même si les serveurs aident à résoudre de plus en plus de problèmes par des remontées d'alerte.

Bien souvent, par souci d'économie (le chef de projet réduit les délais ou prévoit moins de ressources humaines que prévu), la documentation est rédigée et l'optimisation envisagée après la mise en production. Des experts estiment à 60 % le gain potentiel de performances rien que sur l'écriture du code SQL. Sachant que pour certaines applications, pour des raisons de pseudo-portabilité, l'utilisation des procédures cataloguées est proscrite ! Tout devant être codé dans les applications, la marge de manœuvre est ainsi réduite. Par ailleurs, bon nombre de problèmes proviennent du modèle de données et il est souvent trop tard pour modifier la structure de la base.

Les performances ne peuvent être considérées sans un contexte : le disque, la mémoire, les processeurs, le réseau sont autant d'éléments qui entrent en compte lors des mesures. Ainsi, la performance n'a souvent de sens que si elle est associée à une action (on parle alors de *benchmark*) telle qu'une migration

vers une version supérieure, une migration de données d'un *tablespace* à l'autre, après ajout de RAM, changement de disque, etc.

Les acteurs

Plusieurs acteurs influent sur les performances :

- Le concepteur se doit de fournir un modèle conceptuel de qualité, une architecture logicielle raisonnée et une programmation modulaire.
- Le développeur vérifie en principe le modèle relationnel (normalisation et dénormalisation raisonnée), écrit les instructions et les requêtes d'une manière concise. Il programme ses transactions en utilisant le plus possible des procédures cataloguées.
- L'administrateur surveille l'exécution des sessions, organise au mieux les espaces logiques et physiques des bases de données. Il dimensionne la mémoire pour les données et les traitements.
- L'utilisateur final qui se fait toujours connaître quand une attente est trop importante. Il convient de le sensibiliser en amont pour éviter parfois des conflits inutiles.

Contexte et objectifs

Idéalement, l'optimisation doit faire partie du cycle de développement et être mise en œuvre avant la mise en production. Cependant, ce n'est pas toujours le cas, ce qui amène un certain nombre de freins :

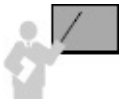
- De quels droits dispose-t-on pour diagnostiquer et identifier le problème ?
- Est-il possible de modifier le schéma relationnel, le code SQL ou applicatif, l'organisation des données (index, types de tables, etc.), la configuration de l'instance, du réseau et du matériel ?

L'organisation des données peut ne pas nécessiter de recompilation (ajout d'index). La solution de changer de prime abord le matériel est souvent une fuite en avant qui peut même s'avérer plus pénalisante avec une machine plus puissante.

L'objectif de toute optimisation doit être précis et mesurable (par exemple, atteindre 90 ms pour extraire la liste des produits d'une commande). En effet,

un objectif flou ne sera jamais satisfait et c'est une garantie contre la tentation d'optimisation excessive et contre-productive.

Utiliser un jeu de tests (comparable aux données en production) permet de mesurer objectivement les performances. Se pose alors le problème de l'accès aux données réelles.



Plus tôt l'optimisation est prise en compte, moins elle sera coûteuse.

Les différents SGBD du marché ne se comportent pas de la même manière (une solution valable pour un SGBD peut se révéler peu performante pour un autre). Cela se vérifie également pour deux versions ou *releases* différentes d'un même SGBD.

Une solution convenable en mode mono-utilisateur peut se révéler non opérationnelle en mode multi-utilisateur.

Causes possibles

Les causes principales d'une mauvaise optimisation des instructions SQL sont :

- les statistiques destinées à l'optimiseur sont obsolètes ;
- des structures d'accès sont inexistantes (index ou partitions) ;
- la sélection de plans d'exécution n'est pas optimale (certains éléments de l'instruction SQL sont mal évalués, par exemple son coût, sa cardinalité ou la sélectivité de son prédicat) ;
- les instructions SQL sont mal construites (conditions de jointure manquantes, mauvais prédicats ou opérateurs, etc.).

Des performances médiocres peuvent également provenir de problèmes matériels (mémoire, entrées-sorties, CPU, disque, etc.).

Présentons maintenant les tables (disponibles en téléchargement) constituant le jeu de tests.

Présentation du jeu d'exemples

Dans ce jeu d'essais, créé initialement par Frédéric Brouard (http://sqlpro.developpez.com/), des adhérents pratiquent différents sports. Deux tables de références (`Adherent` et `Sport`) et une table d'association (`Pratique`) sont mises en œuvre. Les index seront décrits ultérieurement.

La volumétrie initiale de ces tables est la suivante : 24 000 adhérents, 12 sports et plus de 27 000 lignes dans la table `Pratique`. Les index et contraintes référentielles seront définis ultérieurement.

```
CREATE TABLE Sport
 (spid TINYINT UNSIGNED NOT NULL, splibelle VARCHAR(20) NOT NULL);
CREATE TABLE Adherent
 (adhid    SMALLINT UNSIGNED NOT NULL,
  nom      VARCHAR(25) NOT NULL, prenom    VARCHAR(30) NOT NULL,
  civilite VARCHAR(12) NOT NULL, date_nais DATE NOT NULL,
  tel VARCHAR(15));
CREATE TABLE Pratique
  (adhid SMALLINT UNSIGNED NOT NULL, spid TINYINT UNSIGNED NOT
NULL);
```

L'optimiseur

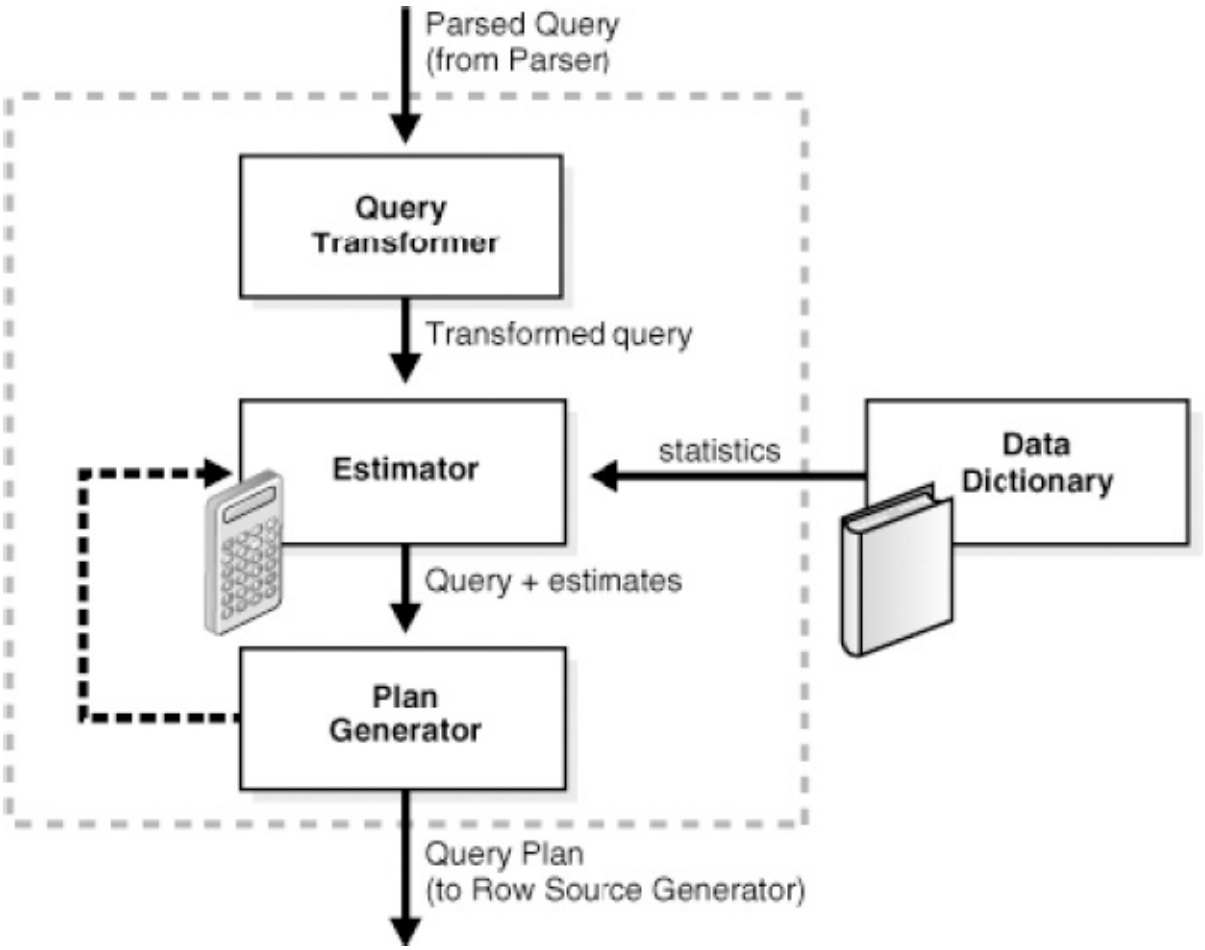
L'optimiseur basé sur les coûts (*cost based optimizer*) estime chaque chemin d'accès en fonction des statistiques disponibles pour les tables concernées. Collecter correctement ces statistiques est donc fondamental.

Fonctionnement

L'optimiseur d'instructions est composé :

- Du transformateur qui dispose en entrée d'une requête *parsée* et la transforme de manière optimale (notamment par la technique d'expansion). Plusieurs plans potentiels sont générés en fonction des chemins d'accès disponibles.
- De l'estimateur qui calcule le coût de chaque requête en fonction des statistiques des tables et des index associés.
- Le générateur qui compare les différents plans et sélectionne celui dont le coût est le plus faible.

***Figure 10-1** Mécanisme de l'optimiseur*



La recherche du meilleur plan d'exécution possible pour une interrogation étant complexe, l'objectif de l'optimiseur est de trouver un « bon » plan, généralement appelé « plan au meilleur coût ». L'optimiseur adapte son plan d'exécution si les statistiques changent. À titre d'exemple, si d'après les statistiques il s'avère que près de la moitié des clients sont de l'Îlede-France, le parcours complet de table (*full scan table*) destiné à extraire les clients de cette région sera plus judicieux que l'utilisation d'un éventuel index.

Expansions

En fonction des index existants, des partitions et des statistiques, et avant d'effectuer un calcul de coût, l'optimiseur peut décider de transformer une requête en une autre équivalente avant de calculer le coût de cette dernière et de l'exécuter. Le principal objectif du transformateur est de déterminer s'il est avantageux de modifier l'instruction afin qu'elle permette la génération du meilleur plan.

Le transformateur utilise plusieurs techniques, comme la transitivité, la fusion de vues, l'inclusion automatique de prédicats, l'extraction de sous-interrogations, la réécriture de requête, la transformation en étoile et l'expansion de l'opérateur `OR`. Le tableau suivant présente quelques équivalences classiques.

Tableau 10-1 Expansions classiques

Expansion	Requête initiale	Requête transformée

Opérateur `OR`	`SELECT adhid,nom,tel`
	`  FROM Adherent`
	`  WHERE civilite = 'Mlle.'`
	`  OR nom = 'LEBLANC';`
		`SELECT adhid,nom,tel FROM Adherent`
		`  WHERE nom = 'LEBLANC'`
		`UNION ALL`
		`SELECT adhid,nom,tel FROM Adherent`
		`  WHERE civilite = 'Mlle.'`
		`  AND nom <> 'LEBLANC';`

Sous-interrogation	`SELECT adhid`
	`  FROM Pratique`
	`  WHERE spid IN`
	`  (SELECT spid FROM Sport`
	`  WHERE splibelle='Tennis');`
		`SELECT Pratique.adhid`
		`  FROM` **`Pratique, Sport`**
		`  WHERE` **`Pratique.spid = Sport.spid`**
		`  AND  Sport.splibelle='Tennis';`

Fusion de vues	`CREATE VIEW Adherent_miss`
	`AS`
	`  SELECT adhid,prenom,`
	`  nom,tel,date_nais`
	`  FROM Adherent`
	`  WHERE civilite = 'Mlle.';`
	`SELECT prenom,nom,tel`
	`  FROM Adherent_miss`
	`  WHERE adhid > 7800;`
		`SELECT prenom,nom,tel`
		`  FROM Adherent`
		`  WHERE` **`civilite = 'Mlle.'`**
		`  AND adhid > 7800;`

Transitivité	`SELECT  p.adhid,s.splibelle`
	`  FROM  Pratique p, Sport s`
	`  WHERE p.spid = s.spid`
	`  AND  s.spid = 12;`
		`SELECT  p.adhid,s.splibelle`
		`  FROM  Pratique p, Sport s`
		`  WHERE p.spid = s.spid`
		`  AND  s.spid = 12`
		`  AND  p.spid = 12;`

L'estimateur

L'estimateur gère trois types de mesures : la sélectivité, la cardinalité et le coût. Ces mesures sont liées entre elles. La cardinalité dérive de la sélectivité et le coût dépend souvent de la cardinalité.

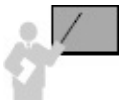
- La sélectivité est une estimation de la proportion des lignes d'un ensemble qui est extraite par un prédicat donné ou une combinaison de prédicats (*1/nombre de valeurs distinctes non nulles*). Le calcul de la sélectivité est basé sur les statistiques. C'est avec la sélectivité que l'optimiseur évalue le nombre d'enregistrements traités pour toute équijointure (`a.adhid=p.adhid`) et prédicat d'égalité (`a.adhid=6`). Plus une colonne est sélective, moins

l'optimiseur envisage de retourner des lignes à l'évaluation du prédicat. La sélectivité forme une partie importante de l'équation décidant du meilleur chemin.

- La cardinalité d'une opération du plan d'exécution d'une requête représente l'estimation du nombre de lignes extraites par cette opération. Généralement, la source est une table, une vue ou encore le résultat d'une jointure ou d'un opérateur `GROUP BY`. Cette valeur est essentielle pour déterminer le coût des opérations de jointure, de filtre et de tri. Pour chaque colonne, on trouve la relation : *cardinalité* = *sélectivité* × *nbre_total_lignes*.

Considérons à titre d'exemple la requête suivante :
```
SELECT adhid, nom FROM Adherent WHERE prenom = 'CELINE';
```

Étant donné que la table contient 24 035 lignes incluant 3 007 prénoms distincts, les indicateurs que l'optimiseur consultera sont les suivants : *sélectivité* = 1/3007 ⇒ 0,00033 ; *cardinalité* = 24035 × 1/3007 ⇒ 7,99.



Le choix d'utiliser tel ou tel parcours (entre les tables pour une jointure par exemple), tel index ou telle partition est déterminé en majeure partie par les valeurs calculées ou estimées de sélectivité et de cardinalité.

Les statistiques destinées à l'optimiseur

Les statistiques sont créées par MySQL dans le but d'améliorer l'action de l'optimiseur. Elles consignent différentes informations concernant le système (utilisation de la CPU et des entrées-sorties), les tables (volumétrie, taille moyenne des lignes, blocs, etc.), les index (clés, nombre de blocs feuilles, etc.), les colonnes (nombre de valeurs distinctes, nombre de `NULL`, taille moyenne) et les données de la table (valeur minimale, valeur maximale et distribution des valeurs).

Toutes ces informations vont permettre de choisir parmi les algorithmes disponibles pour générer chaque plan d'exécution. L'optimiseur choisira, pour une requête donnée, le plan d'exécution le moins coûteux.

L'optimiseur considère par défaut que les données de toute colonne sont

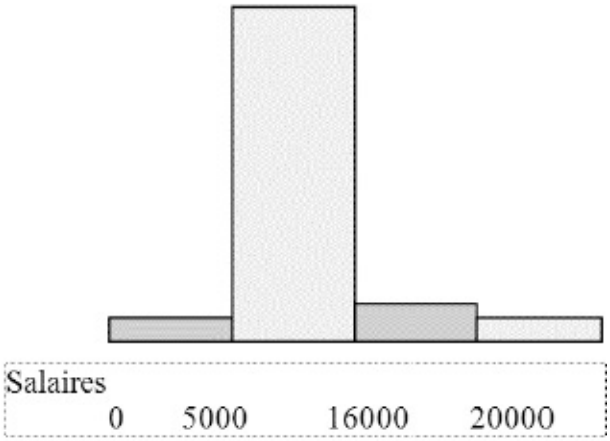
réparties de façon uniforme (hypothèse de distribution uniforme des valeurs). Ce comportement peut entraîner la génération de plans d'exécution non optimaux en cas de répartition inégale des données. Les statistiques se doivent donc de refléter au mieux le contenu des tables.

Considérons la table suivante contenant 1 000 lignes. Si MySQL collecte trois valeurs distinctes pour la colonne compa (on suppose qu'il n'existe que trois compagnies), alors les plans d'exécution générés considéreront cette répartition (pour la recherche des pilotes d'une compagnie donnée, MySQL s'attendra à monter l'équivalent de 333 lignes en mémoire). En revanche, si le salaire se répartit inégalement (la majorité de pilotes gagnant entre 5 000 et 16 000 €), MySQL générera un histogramme plus précis pour que l'optimiseur ne se base pas sur une répartition homogène et prévoit moins de pilotes à extraire pour des faibles ou des hauts salaires.

Figure 10-2 *Répartition des données dans une colonne*

Pilote

brevet	salaire	*compa*
PL-1	3400	AF
PL-2	4500	SING
PL-3	9000	AF
PL-4	10000	SING
PL-5	10050	SING
PL-6	16000	AF
PL-7	10060	CAST
PL-8	15000	
...	...	...
PL-1000		CAST



Collecte

La commande `ANALYZE TABLE` collecte les statistiques sur une ou plusieurs tables `MyISAM` ou `InnoDB`. L'option `NO_WRITE_TO_BINLOG` (ou son alias `LOCAL`) convient aux architectures qui ne sont pas en *cluster* (pas de mise à jour du fichier de journalisation).

Vous devez détenir les privilèges `SELECT` et `INSERT` sur la ou les tables concernées. La syntaxe de cette commande est la suivante :

```
ANALYZE [NO_WRITE_TO_BINLOG | LOCAL] TABLE
        nom_table [, nom_table] ...
```

Pendant la collecte, les tables sont verrouillées en lecture. MySQL retourne plusieurs informations sous la forme d'un résultat d'une requête. Le tableau suivant détaille la collecte des statistiques sur les trois tables de notre exemple (situées dans la base `bdsoutou`).

Tableau 10-2 Collecte des statistiques sur une table

Collecte des statistiques	Commentaires

`mysql> ANALYZE NO_WRITE_TO_BINLOG TABLE Adherent, Pratique, Sport;`
`+-------------------+---------+----------+----------+`
`| Table | Op | Msg_type | Msg_text |`
`+-------------------+---------+----------+----------+`
`| bdsoutou.adherent | analyze | status | OK |`
`| bdsoutou.pratique | analyze | status | OK |`
`| bdsoutou.sport | analyze | status | OK |`
`+-------------------+---------+----------+----------+`
	`Table` : nom de la table analysée.
	`Op` : toujours `analyze`.
	`Msg_type` : une des valeurs `status`, `error`, `info`, `note` ou `warning`.
	`Msg_text` : message d'information.

Cette procédure est utile pour les traitements manipulant des tables temporaires que les statistiques automatiques peuvent ignorer et qui sont privées d'index.

Après avoir lancé cette procédure, il est possible d'examiner quelques résultats provenant de la distribution des données. Pour les colonnes non indexées, la commande `PROCEDURE ANALYSE()` est à utiliser ; pour les colonnes indexées, vous devrez opter pour `SHOW INDEX FROM` *`nom_table`* ou interroger la table `INFORMATION SCHEMA STATISTICS` en selectionnant la table.

Visualisation des statistiques

La clause `PROCEDURE ANALYSE()` utilisée dans une instruction `SELECT` permet de préconiser les types les plus adéquats pour chaque colonne (surtout pour les numériques). La syntaxe de cette clause est la suivante :

```
SELECT ... FROM ...
[WHERE ... ]
```

```
PROCEDURE ANALYSE([max_elements,[max_memory]]);
```

- *max_elements* (par défaut, 256) indique le nombre maximal de valeurs distinctes que l'analyse doit considérer par colonne. Par exemple, ce nombre permet de préconiser ou non un type ENUM (si la cardinalité d'une colonne dépasse ce nombre, ce type ne sera pas suggéré).
- *max_memory* (par défaut, 8192) est la capacité maximale de mémoire que l'analyse alloue par colonne.

Le tableau suivant présente le détail des statistiques concernant la table des adhérents. MySQL retourne les informations sous la forme de colonnes. Pour chaque colonne analysée, les différents champs sont :

- Field_name qui désigne le nom de la colonne.
- Min_value qui désigne la plus petite valeur stockée en base.
- Max_value qui désigne la plus grande valeur stockée en base.
- Min_length qui désigne la plus petite taille des valeurs stockées en base.
- Max_length qui désigne la plus grande taille des valeurs stockées en base.
- Empties_or_zeros qui désigne le nombre de valeurs vides (pour les caractères) ou nulles (pour les numériques).
- Nulls qui désigne le nombre de valeurs NULL.
- Avg_value_or_avg_length qui désigne la moyenne des valeurs (ou des tailles dans le cas des chaînes de caractères et des dates).
- Std qui désigne l'écart type pour les numériques (NULL dans le cas des chaînes de caractères et des dates).
- Optimal_fieldtype qui propose le type le plus adéquat.

Tableau 10-3 Informations sur les colonnes

Requêtes et résultats

```
mysql> SELECT adhid, nom, prenom, civilite, date_nais, tel
      FROM Adherent
      PROCEDURE ANALYSE(10);

+--------------------+---------+-------+-----------------+-------------+-----------------
--------+
| Field_name         | Min_value       | Max_value       | Min_length  |
Max_length            |
| Empties_or_zeros   | Nulls   | Avg_value_or_avg_length | Std         |
```

```
Optimal_fieldtype       |
+--------------------+---------+------------------------+-------------+----------------
--------+
| Adherent.adhid     | 1              | 26122          |           1
|                      5 |
|                  0 |       0 | 13147.9308             |   7544.6148 |
SMALLINT(5)             |
|                                                                        UNSIGNED NOT
NULL       |
| Adherent.nom       | AABADLI         | ZYSSMAN         |           1
|                      25 |
|                  0 |       0 | 7.0870                 |        NULL | VARCHAR(25)
NOT NULL    |
| Adherent.prenom    | A MARIE         | ZUONKO          |           2
|                      18 |
|                0   |       0 | 6.8695                 |        NULL | VARCHAR(18) NOT
NULL    |
| Adherent.civilite  | Mlle.           | Mr.             |           3
|                      5 |
|                  0 |       0 | 3.7462                 |        NULL |
ENUM('Mlle.','Mme.',    |
|                                                                             'Mr.') NOT
NULL          |
| Adherent.date_nais | 1905-04-20      | 2009-07-26      |          10
|                      10 |
|                  0 |       0 | 10.0000                |        NULL | DATE NOT
NULL           |
| Adherent.tel       |  01-00-00-20-42 |  08-88-88-85-46 |          15
|                      15 |
|                  0 |     765 | 13.9993                |        NULL |
TINYTEXT                |
+--------------------+---------+------------------------+-------------+----------------
--------+
```

Les types que MySQL préconise sont à comparer avec le type de chaque colonne de la table. Force est de constater qu'il suffisait effectivement de déclarer la colonne `civilite` en tant qu'énumération. Seule la colonne `tel` comporte des `NULL` (765 adhérents n'ont pas de téléphone).

Dans le même ordre d'idée, vous pouvez aussi consulter la section *Choosing the Right Type for a Column* de la documentation, disponible à l'adresse suivante : http://dev.mysql.com/doc/refman/x.y/en/choosing-types.html.

Quand mettre à jour les statistiques ?

La meilleure fréquence d'actualisation des statistiques est « dès que nécessaire ! ». Concernant les tables `MyISAM`, les statistiques sont mises à jour automatiquement (et dès `INSERT` ou `DELETE`). Concernant les tables `InnoDB`, les statistiques sont mises à jour dès qu'un seizième des données d'une colonne est modifié (6,25 %).

Des statistiques obsolètes nuisent au plan d'exécution et peuvent provoquer de

réels écarts de performances. Il est courant de les collecter périodiquement pour des bases en production.

Dans tous les cas, l'actualisation des statistiques s'impose après des modifications fréquentes et significatives au cours de la journée, une migration, une importation conséquente ou une modification du modèle physique (changement d'un paramètre de stockage, création d'index, partitionnement, réorganisation, etc.).

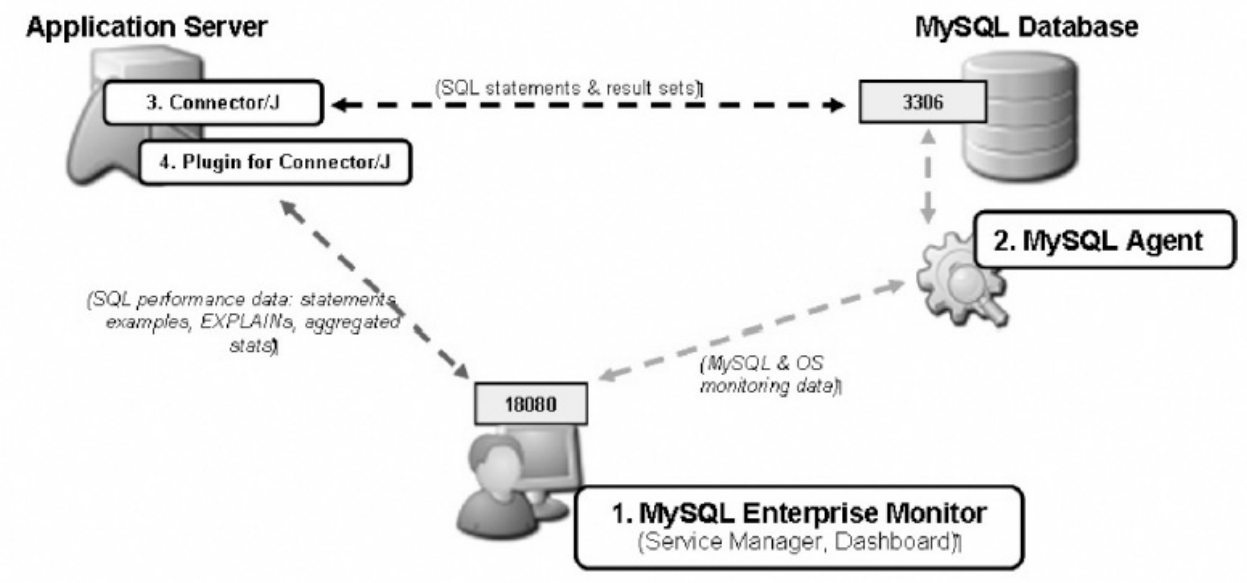
Outils de mesure de performances

Cette section présente les différentes méthodes qui permettent d'évaluer les performances des requêtes. Une fois les problèmes identifiés, il conviendra d'élaborer une stratégie en fonction des outils à disposition (création d'index, réécriture de la requête, utilisation de tables temporaires, partitionnement ou modification du schéma).

MySQL Query Analyzer

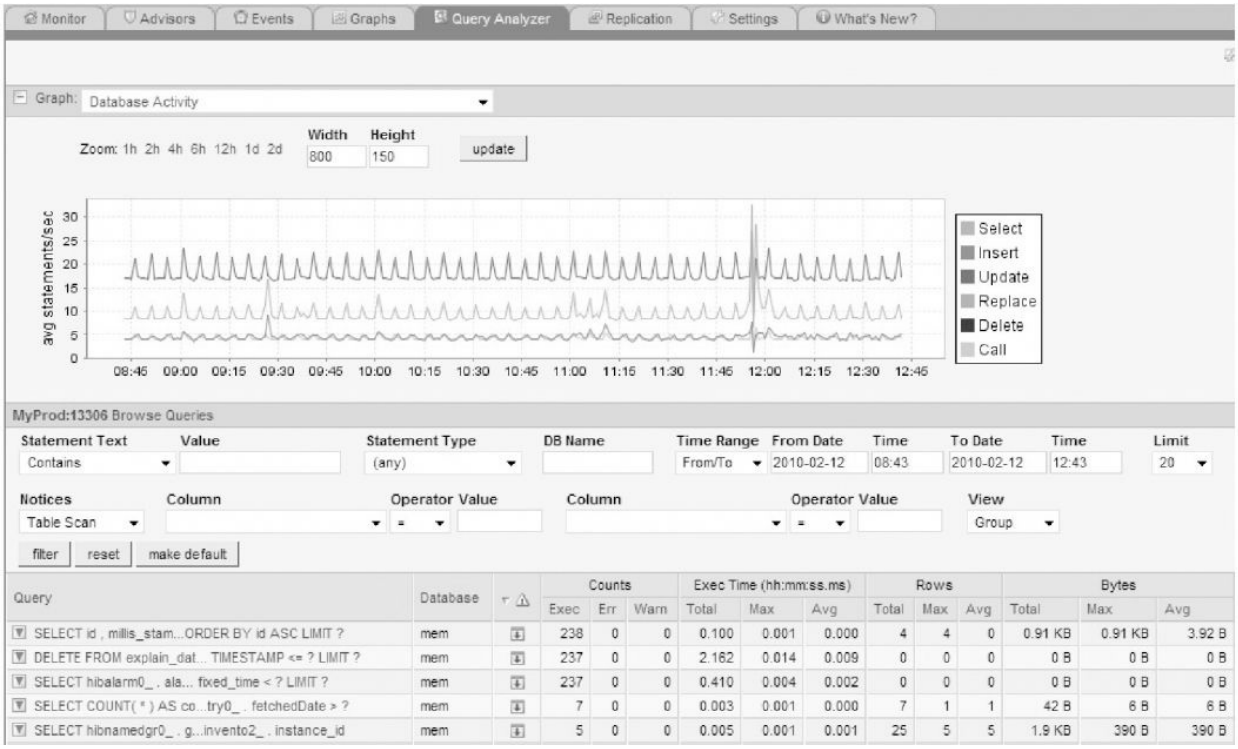
Reportez-vous aux compléments Web pour le récapitulatif des possibilités offertes par la console MySQL Enterprise, qui inclut le module MySQL Query Analyser.

Figure 10-3 *Offre MySQL Enterprise © Document MySQL*



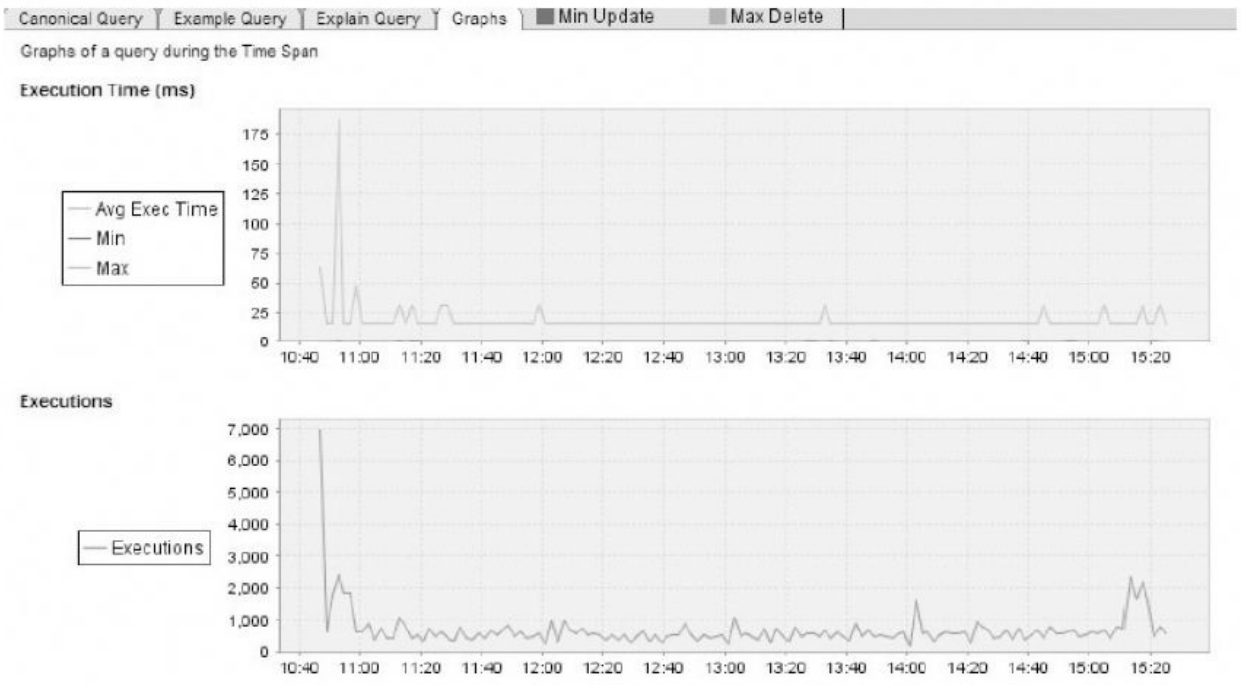
L'affichage centralise les instructions qui se déroulent sur un serveur. Plusieurs indicateurs permettent d'isoler facilement les contentions comme le nombre total d'exécutions, le nombre d'erreurs et de *warnings*, les temps d'exécution et la volumétrie mise en œuvre.

Figure 10-4 *MySQL Query Analyzer*



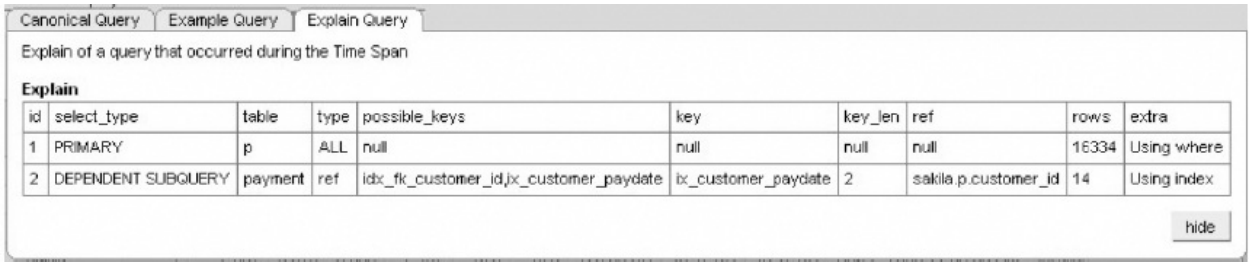
Il est possible d'étudier en détail une des instructions et de visualiser sous forme de graphes certains indicateurs.

Figure 10-5 *Détails d'une requête*



Le plan d'exécution est présenté sous une forme tabulaire.

Figure 10-6 *Plan d'exécution d'une requête*



id	select_type	table	type	possible_keys	key	key_len	ref	rows	extra
1	PRIMARY	p	ALL	null	null	null	null	16334	Using where
2	DEPENDENT SUBQUERY	payment	ref	idx_fk_customer_id,ix_customer_paydate	ix_customer_paydate	2	sakila.p.customer_id	14	Using index

Visualisation des plans d'exécution

Un plan d'exécution est le résultat de l'action de l'optimiseur qui présente au moteur les opérations qu'il va effectuer pour mettre en œuvre une requête. Pour visualiser un plan d'exécution, vous devrez utiliser la commande `EXPLAIN`.

MySQL 5.6 et ultérieures

Depuis la version 5.6.3, la commande EXPLAIN permet de tracer les instructions SELECT, DELETE, INSERT, REPLACE et UPDATE (avant la version 5.6.3, seule l'instruction SELECT était prise en compte dans l'affichage d'un plan d'exécution).

La commande EXPLAIN retourne une ligne pour chaque table interrogée. Les tables sont listées dans l'ordre dans lequel elles sont lues au cours de la requête.

```
EXPLAIN [EXTENDED | PARTITIONS]
 {SELECT… | DELETE… | INSERT… |  REPLACE… | UPDATE… };
```

L'option EXTENDED fournit des informations supplémentaires (notamment sur la façon dont l'optimiseur réécrit la requête) qui peuvent être affichées par SHOW WARNINGS. L'option PARTITIONS concerne les tables partitionnées (voir plus loin).

L'exemple suivant présente le plan d'exécution d'une jointure entre trois tables afin d'extraire l'identité des joueurs de handball nés en 1995. Il apparaît que les tables sont parcourues sans index (*full scan*). MySQL choisit d'extraire le sport, puis réalise une jointure avec la table d'association et une autre jointure avec la table des adhérents.

Tableau 10-4 Plan d'exécution d'une jointure relationnelle

Requête et plan d'exécution

```
mysql> EXPLAIN
     SELECT  a.adhid, a.prenom, a.nom
      FROM   Adherent a, Pratique p, Sport s
       WHERE EXTRACT(YEAR FROM a.date_nais) = 1995
       AND   a.adhid     = p.adhid
       AND   s.spid      = p.spid
       AND   s.splibelle = 'Handball' ORDER BY nom;
+----+-------------+-------+------+---------------+------+---------+------+-------+------
-----------+
| id | select_type | table | type | possible_keys | key  | key_len | ref  | rows  |
Extra          |
+----+-------------+-------+------+---------------+------+---------+------+-------+------
-----------+
|  1 | SIMPLE      | s     | ALL  | NULL          | NULL | NULL    | NULL |    12
|Using where;
                                                                                  Using
temporary;
                                                                                  Using
filesort   |
```

```
|  1 | SIMPLE      | p     | ALL  | NULL          | NULL | NULL    | NULL | 27651
|Using where;
                                                                                Using
join buffer|
|  1 | SIMPLE      | a     | ALL  | NULL          | NULL | NULL    | NULL | 24243
|Using where;
                                                                                Using
join buffer|
+----+-------------+-------+------+---------------+------+---------+------+-------+------
-----------+
```

Les colonnes retournées sont les suivantes :

- `id` désigne l'identifiant de la requête (numérotation allant de 1 à *n* si plusieurs requêtes sont mises en œuvre dans l'extraction).
- `select_type` désigne le type de chaque extraction (`SELECT` principal ou imbriqué).

Tableau 10-5 Significations de `select_type`

Valeurs	Significations
`SIMPLE`	Requête simple (sans `UNION` ni sous-requête).
`PRIMARY`	Requête principale (dans le cas de sous-requêtes).
`UNION`	Requête suivante dans le cas d'une union.
`DEPENDENT UNION`	Requête suivante dans le cas d'une union dépendant d'une requête extérieure.
`UNION RESULT`	Résultat d'une union.
`SUBQUERY`	Premier `SELECT` dans une sous-requête.
`DEPENDENT SUBQUERY`	Premier `SELECT` dans une sous-requête dépendant d'une requête extérieure.
`DERIVED`	Table dérivée (sous-requête d'une clause `FROM`).
`UNCACHEABLE SUBQUERY`	Sous-requête dont le résultat ne peut être mis en cache et doit être réévalué à chaque ligne de la requête extérieure.
`UNCACHEABLE UNION`	Deuxième (ou suivant) `SELECT` dans une union qui correspond à une sous-requête du type précédent.

- `table` désigne la table concernée.

- `type` désigne la stratégie d'accès aux données choisie par l'optimiseur. Le tableau suivant présente les différentes stratégies (de la plus efficace à la moins optimale) avec les cas d'usages associés les plus courants.

Tableau 10-6 Significations de `type`

Valeurs	Significations
`system`	La table ne dispose que d'une ligne en mémoire (table système). Cas particulier du type suivant.
`const`	La table dispose au moins d'une ligne à extraire lors de la comparaison d'une clé primaire ou `UNIQUE` à une constante. En regard de cette valeur, l'enregistrement est constant pour l'optimiseur et n'est ainsi exploité qu'une seule fois.
`eq_ref`	La table dispose au moins d'une ligne à extraire lors d'une équijointure (via une clé primaire ou un index `UNIQUE NOT NULL`).
`ref`	Concerne les équijointures via une clé primaire, un index `UNIQUE` ou un index partiellement renseigné en partie gauche (*leftmost prefix*). Les inéquijointures (basées sur l'inégalité `<>`) sont également concernées.
`fulltext`	Jointures sur des colonnes indexées `FULLTEXT`.
`ref_or_null`	Identique à `ref` en incluant les recherches sur les colonnes contenant des valeurs `NULL`.
`index_merge`	Cette optimisation consiste à fusionner le résultat de l'extraction de plusieurs index.
`unique_subquery`	Remplace l'opérateur `ref` pour les sous-requêtes `IN` retournant une seule ligne (exemple des clés primaires ou index uniques et non nuls).
`index_subquery`	Similaire au précédent mais ne concerne que les sous-requêtes `IN` retournant plusieurs lignes par l'intermédiaire de colonnes indexées non nulles (et non uniques).
	L'index est parcouru sur un intervalle de valeurs (*index range scan*). La colonne indexée est indiquée dans le champ `key` et le champ `ref` vaut `NULL`. Cette

`range`	opération se retrouve suite à des comparaisons entre clés et constantes par l'intermédiaire de `=`, `<>`, `>`, `>=`, `<`, `<=`, `IS NULL`, `<=>`, `BETWEEN`, et `IN(…)`.
`index`	L'index est parcouru tout entier (*index full scan*). Cette opération est en principe plus rapide que la suivante du fait de la taille réduite d'un index par rapport à celle de la table. Ce traitement peut intervenir à l'extraction de colonnes constituant un index.
`ALL`	La table est parcourue en entier (*full table scan*). Cette opération est en principe à éviter du fait de la taille importante que peut avoir une table. L'optimiseur choisit à raison cette stratégie dans certains cas, même en présence d'index.

- `possible_keys` liste les index potentiellement utilisables. Cette information est indépendante de l'ordre d'affichage des tables. Quand cette colonne vaut `NULL`, il n'y a pas d'indexation prévue (c'est peut être le moment d'ajouter un nouvel index).
- `key` indique l'index choisi par l'optimiseur. Il est possible qu'il ne figure pas dans la liste précédente (cas des index couvrants). Pour les tables `InnoDB`, un index secondaire peut remplacer une clé primaire. Quand cette colonne vaut `NULL`, il n'y a pas d'indexation possible (c'est sans doute le moment d'agir).
- `key_len` indique la taille en octets de l'index choisi. Cette information permet de déterminer si l'optimiseur utilise tout ou partie d'un index composite.
- `ref` indique la (les) colonne(s) ou la (les) constante(s) choisie(s) pour être comparée(s) à l'index (de la colonne `key`).
- `rows` indique le nombre de lignes estimé par l'optimiseur.
- `filtered` indique le pourcentage estimé de lignes extraites par la condition. Ainsi, l'expression *rows*× *filtered*/100 correspond au nombre de lignes qui seront jointes aux tables précédentes. La colonne `filtered` est affichée avec l'option `EXTENDED`.
- `Extra` renseigne l'implémentation de la requête. Le tableau suivant présente les principales informations avec des cas d'usages associés les plus courants. Les statégies les plus pénalisantes sont `Using filesort` et `Using`

temporary.

Tableau 10-7 Significations de `Extra`

Valeurs	Significations
`const row not found`	Dans une requête `SELECT… FROM t`, la table `t` est vide.
`Distinct`	L'optimiseur recherche des valeurs distinctes, et dès la première trouvée pour l'enregistrement courant, cette recherche cesse.
`Full scan on NULL key`	Optimisation d'une sous-requête si aucun index ne peut être exploité.
`Impossible HAVING`	La clause `HAVING` retourne toujours *false* et l'extraction ne sélectionne aucune ligne.
`Impossible WHERE`	La clause `WHERE` retourne toujours *false* et l'extraction ne sélectionne aucune ligne.
`No matching min/max row`	Aucune ligne ne satisfait une requête `SELECT MIN(…)FROM...`
`no matching row in const table`	La jointure concerne une table vide ou une table dans laquelle aucune ligne ne vérifie la condition sur un index unique.
`Range checked for each record (index map: masque)`	L'optimiseur ne trouve pas d'index satisfaisant et effectue une opération de fusion (*index merge*). Les index sont numérotés de 1 à *n* dans la commande `SHOW INDEX`. Le masque codifie les index utilisés dans la fusion. Par exemple, 11001 signifie que les index numérotés 1, 2 et 5 seront considérés.
`unique row not found`	Pour une requête `SELECT… FROM t`, aucune ligne ne satisfait la condition sur un index unique ou une clé primaire.
`Using filesort`	L'optimiseur effectue un traitement additionnel pour trier des lignes.
`Using index`	L'extraction et le test ne concernent que l'index et pas la ou les lignes associées.

`Using index for group-by`	Un index est trouvé pour traiter toutes les colonnes d'une requête `GROUP BY` ou `DISTINCT`.
`Using join buffer`	Les lignes de précédentes jointures sont positionnées dans le tampon *join buffer* et sont utilisées par la jointure courante.

`Using sort_union(...),`
`Using union(...),`
`Using intersect(...)`
	Indique comment les index sont parcourus lors de la fusion (*index merge*).

`Using temporary`	L'optimiseur a besoin de créer une table temporaire, généralement lors de `GROUP BY` et `ORDER BY` complexes.
`Using where`	Une clause `WHERE` est utilisée. À moins que vous n'ayez besoin de parcourir toute la table, il se peut que la requête soit mal écrite si la stratégie est `ALL` ou `index` et sans l'existence de `Using where`.

L'exemple suivant présente le plan d'exécution restreint de la même jointure écrite sous la forme procédurale. La requête principale est notée `PRIMARY`, c'est sur elle que portera le tri final. Les tables sont toujours parcourues sans index (*full scan*) du fait de leur absence.

Tableau 10-8 Plan d'exécution d'une jointure procédurale

Requête et plan d'exécution

```
mysql> EXPLAIN EXTENDED
       SELECT adhid, prenom, nom FROM Adherent WHERE adhid IN
              (SELECT adhid FROM Pratique WHERE  spid IN
                     (SELECT spid FROM Sport WHERE splibelle = 'Handball'))
              AND EXTRACT(YEAR FROM date_nais) = 1995 ORDER BY nom;
+----+--------------------+----------+------+-------+----------+-------------------------
----+
| id | select_type        | table    | type | rows  | filtered |
Extra                        |
+----+--------------------+----------+------+-------+----------+-------------------------
----+
|  1 | PRIMARY            | Adherent | ALL  | 24322 |   100.00 | Using where; Using
filesort |
|  2 | DEPENDENT SUBQUERY | Pratique | ALL  | 25513 |   100.00 | Using
where                |
|  3 | DEPENDENT SUBQUERY | Sport    | ALL  |    12 |   100.00 | Using
where                |
+----+--------------------+----------+------+-------+----------+-------------------------
----+
```

La traduction de cette requête est obtenue par la commande `SHOW WARNINGS`.

Tableau 10-9 Traduction de la requête

Commande et résultat

```
mysql> SHOW WARNINGS;
+-------------------------------------------------------------------------------------------
--+
| Level | Code |
Message                                                                          |
+-------------------------------------------------------------------------------------------
--+
| Note  | 1003 | select `bdsoutou`.`adherent`.`adhid` AS `adhid`,
                        `bdsoutou`.`adherent`.`prenom` AS `prenom`,
                        `bdsoutou`.`adherent`.`nom` AS `nom`
          from `bdsoutou`.`adherent`
          where (<in_optimizer>(`bdsoutou`.`adherent`.`adhid`,
                 <exists>(select 1 from `bdsoutou`.`pratique`
                           where (<in_optimizer>(`bdsoutou`.`pratique`.`spid`,
                  <exists>(select 1 from `bdsoutou`.`sport`
                            where ((`bdsoutou`.`sport`.`splibelle`='Handball')
                  and (<cache>
(`bdsoutou`.`pratique`.`spid`)=`bdsoutou`.`sport`.`spid`))))
          and (<cache>(`bdsoutou`.`adherent`.`adhid`) = `bdsoutou`.`pratique`.`adhid`))))
          and (extract(year from `bdsoutou`.`adherent`.`date_nais`) = 1995))
          order by
`bdsoutou`.`adherent`.`nom`                                                       |
+-------------------------------------------------------------------------------------------
--+
```

Organisation des données

Cette section décrit les mécanismes qui vous permettront d'optimiser vos applications. Ils peuvent être conjointement mis en œuvre, il s'agit des contraintes, des index, des tables temporaires et du partitionnement.

Les contraintes

Plus vous définirez de contraintes sur vos colonnes, mieux l'optimiseur sera renseigné. Bien que la contrainte `CHECK` ne soit pas encore utilisée, ni par le SGBD, ni par l'optimiseur, il est possible que cette fonctionnalité soit implémentée dans les prochaines versions.

Les colonnes `NOT NULL`

Déclarez des contraintes `NOT NULL` ne vous dispense pas de réaliser des tests dans les programmes d'application. En effet, il peut être utile de vérifier qu'une valeur est présente dans un champ de saisie d'un formulaire plutôt que d'attendre d'envoyer un grand nombre d'octets au serveur qui renverra une

erreur du fait d'un NOT NULL.

En supposant qu'on ajoute la colonne federation à la table Sport, le tableau suivant présente la déclaration d'une nouvelle contrainte NOT NULL.

Tableau 10-10 Déclaration de NOT NULL

Déclaration en ligne (*in line*)

```
ALTER  TABLE Sport ADD federation VARCHAR(10);
UPDATE Sport SET   federation = '';
ALTER  TABLE Sport MODIFY federation VARCHAR(10) NOT NULL;


mysql> DESCRIBE Sport ;
+------------+---------------------+------+-----+---------+-------+
| Field      | Type                | Null | Key | Default | Extra |
+------------+---------------------+------+-----+---------+-------+
| spid       | tinyint(3) unsigned | NO   |     | NULL    |       |
| splibelle  | varchar(20)         | NO   |     | NULL    |       |
| federation | varchar(10)         | NO   |     | NULL    |       |
+------------+---------------------+------+-----+---------+-------+
```

Les colonnes UNIQUE

Pour toute contrainte UNIQUE, un index (unique) est créé. Une contrainte UNIQUE diffère d'une contrainte PRIMARY KEY par le fait que les NULL sont autorisés et qu'elle n'a pas vocation à identifier chaque ligne. Si NOT NULL est ajouté, la ou les colonnes peuvent se substituer à la clé primaire (on parle de « clé candidate ») et des clés étrangères peuvent référencer cette ou ces colonnes.

Définissez la contrainte UNIQUE sur les colonnes à valeurs uniques pour que l'optimiseur puisse bénéficier d'un index supplémentaire. La désactivation d'une contrainte UNIQUE provoque la suppression de l'index.

Du fait qu'il existe des homonymes au sein des adhérents, la contrainte UNIQUE à mettre en œuvre est composée du nom, du prénom et du numéro de téléphone. Le tableau suivant présente la déclaration de la contrainte un_nom_prenom_tel de type UNIQUE et par conséquence la création d'un index.

Les caractéristiques de cet index sont déterminées à l'aide de la commande SHOW INDEX.

Tableau 10-11 Déclaration d'une contrainte `UNIQUE`

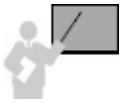
Déclaration de la contrainte

```
ALTER TABLE Adherent
      ADD CONSTRAINT un_nom_prenom_tel UNIQUE (nom,prenom,tel);


mysql> SHOW INDEX FROM Adherent FROM bdsoutou;
+-------------+------------+-------------------+--------------+-------------+
| Table       | Non_unique | Key_name          | Seq_in_index | Column_name |
| Cardinality | Sub_part   | Packed            | Null         | Index_type  |
+-------------+------------+-------------------+--------------+-------------+
|    Adherent |          0 | un_nom_prenom_tel |            1 | nom         |
          198 |       NULL | NULL              |              | BTREE       |
|    Adherent |          0 | un_nom_prenom_tel |            2 | prenom      |
          198 |       NULL | NULL              |              | BTREE       |
|    Adherent |          0 | un_nom_prenom_tel |            3 | tel         |
          198 |       NULL | NULL              | YES          | BTREE       |
+-------------+------------+-------------------+--------------+-------------+
```

Ces caractéristiques sont les suivantes :

- `on_unique` vaut 0 si l'index n'est pas unique (1 s'il l'est).
- `Key_name` correspond au nom de l'index.
- `Seq_in_index` représente la position de la colonne dans l'index (commence à 1).
- `Column_name` indique le nom de la colonne.
- `Cardinality` correspond au nombre estimé de valeurs uniques dans l'index. Cette valeur est inexploitable si l'analyse de la table (`ANALYZE TABLE`) n'a pas encore eu lieu (c'est ici le cas, car plus de 20 000 lignes sont concernées). Plus ce nombre est important, plus l'index risque d'être utilisé.
- `Sub_part` indique le nombre de caractères indexés (si la colonne n'est qu'en partie indexée). Si la colonne est entièrement indexée, `NULL` est affiché.
- `Packed` précise si l'index est compressé (`NULL` sinon).
- `Null` indique si la colonne contient ou pas des valeurs `NULL`.
- `Index_type` indique le type de l'index (`BTREE`, `FULLTEXT`, `HASH` ou `RTREE`).



L'index multicolonne (`nom`+`prenom`+`tel`) profitera aux extractions dont le prédicat est basé sur le nom, le nom et le prénom, le nom et le téléphone ou sur les trois colonnes simultanément.

Le tableau suivant illustre quelques requêtes qui bénéficient ou non de l'index en fonction des colonnes testées.

Tableau 10-12 Utilisation d'un index multicolonne

Index utilisé	Index non utilisé

`SELECT adhid, prenom, tel FROM Adherent`
`WHERE nom = 'THIRIET';`
	`SELECT adhid, nom`
	`FROM Adherent`
	`WHERE prenom = 'JENIFER'`
	`AND tel = '08-11-06-64-14';`

`SELECT adhid, tel FROM Adherent`
`WHERE nom = 'THIRIET'`
`AND prenom = 'JENIFER';`
	`SELECT adhid, prenom, nom`
	`FROM Adherent`
	`WHERE tel = '08-11-06-64-14';`

`SELECT adhid FROM Adherent`
`WHERE nom = 'THIRIET'`
`AND prenom = 'JENIFER'`
`AND tel = '08-11-06-64-14';`	

La désactivation d'une contrainte UNIQUE ne peut se programmer que par la suppression de l'index associé. Ainsi, suite à l'instruction suivante, plus aucune requête ne pourra bénéficier d'une quelconque indexation.

```
ALTER TABLE Adherent
      DROP INDEX un_nom_prenom_tel;
```

Indexation

Les différents types d'index ainsi que la syntaxe de création ont été brièvement présentés au chapitre 1. Nous étudierons ici les index *B-tree*, ceux basés sur une table de hachage (*hash index*) et ceux relatifs aux données textuelles (*fulltext index*). Un autre type d'index intéresse les données géographiques, il s'agit du *R-tree*.

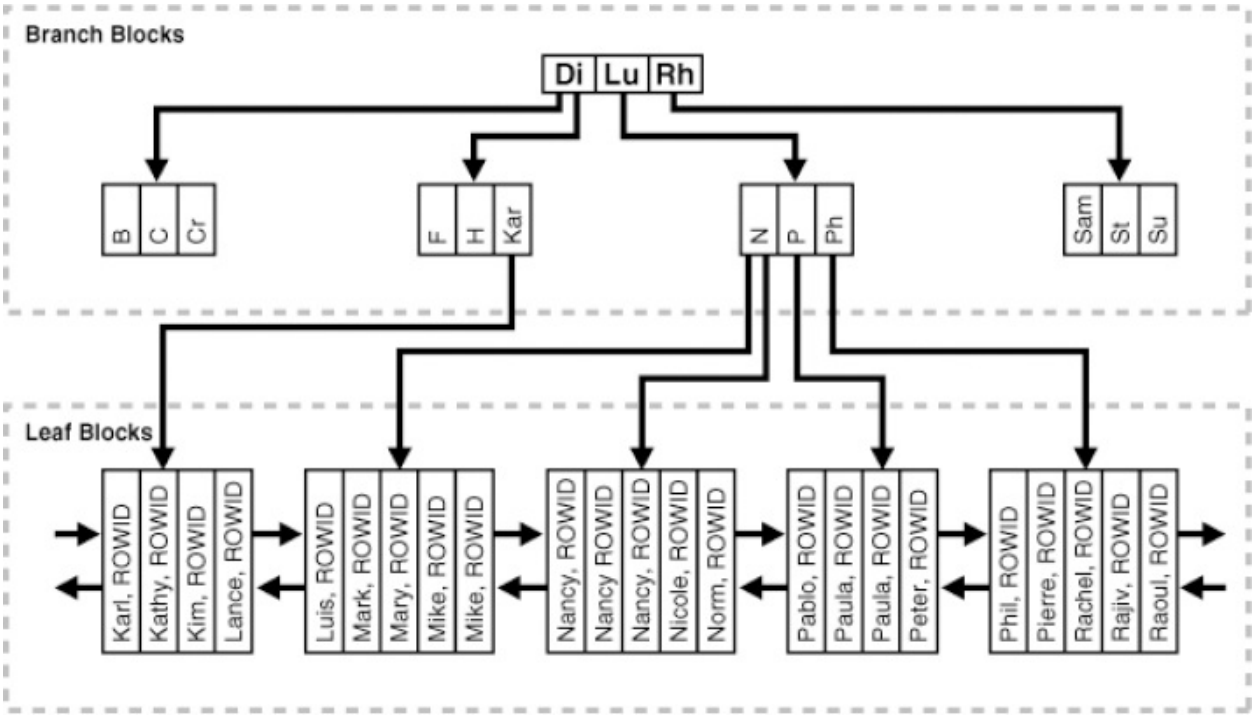
Quel que soit le type d'index considéré, le principe général est toujours le même et consiste à associer rapidement l'adresse d'une ou de plusieurs lignes en fonction d'une valeur d'une ou de plusieurs colonnes constituant l'index. Sans indexation, toute recherche nécessite un parcours séquentiel de toute la table (montée en mémoire de tous les blocs qui contiennent ces lignes). Ce procédé devient très pénalisant dès que le volume de données monte en charge. Ainsi, le nombre d'accès augmente proportionnellement avec le nombre de lignes traitées (une table 1 000 fois plus volumineuse impliquera bien souvent un coût d'accès 1 000 fois plus élevé).

Index B-tree

Les index *B-tree* (*B* comme *Balanced*) sont constitués comme des arbres dont les nœuds aiguillent vers des sous-nœuds (suivant la valeur recherchée) jusqu'aux blocs feuilles (*leaf blocks*) qui contiennent toutes les valeurs de l'index et les adresses des lignes contenues dans les blocs de données associés. En théorie, les blocs feuilles sont doublement chaînés de sorte que l'index puisse être parcouru dans les deux sens, sans nécessairement passer par la racine.

Ce mécanisme d'accès aux données est infiniment plus performant qu'un classique parcours séquentiel. Il s'apparente à une recherche dichotomique. En effet, pour une recherche, le nombre de blocs accédés n'est plus proportionnel à *n*, mais à *log*(*n*) (une table 1 000 fois plus volumineuse ajoute des entrées à l'index sans forcément ajouter un niveau à celui-ci). De plus, ce procédé fonctionne pour des recherches qui intéressent plusieurs blocs, car elle permet au bloc feuille de contenir plusieurs adresses pour une même valeur d'index (par exemple, une recherche sur le nom d'un adhérent qui n'est pas forcément unique dans la table).

Figure 10-7 *Index B-tree © Document Oracle*



Pour les tables `MyISAM` et `InnoDB`, un index *B-tree* est généré automatiquement lors

de la création d'une clé primaire et d'une contrainte `UNIQUE`. Les index des tables `MyISAM` ne sont pas chaînés au niveau des feuilles de l'arbre. Pour les tables associées au moteur `MEMORY`
(dans ce cas, données et index ne résident qu'en mémoire), l'index généré par défaut est un *hash index* (mécanisme étudié par la suite).

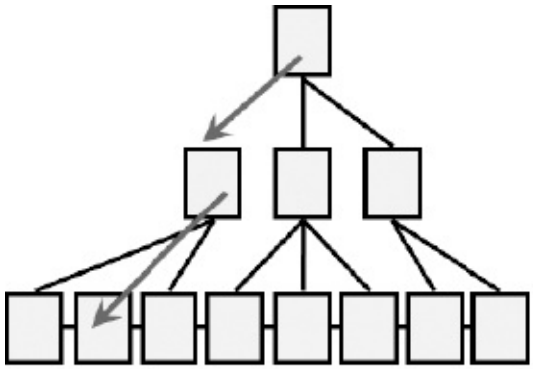
Les arbres *B-tree* présentent de nombreux avantages :

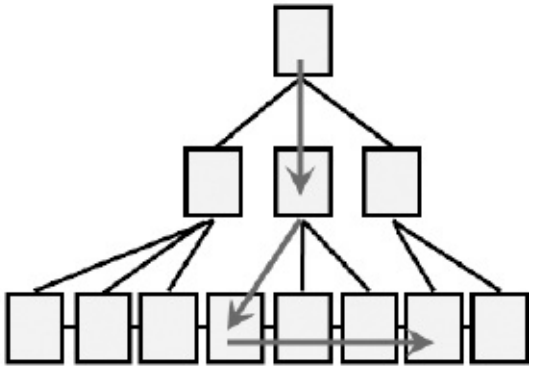
- Malgré les mises à jour de la table, ils restent équilibrés (les blocs feuilles sont au même niveau). En conséquence, quelle que soit la valeur cherchée, le temps de parcours est sensiblement identique. Les blocs intermédiaires sont remplis, en moyenne, au trois quart de leur capacité.
- Excellentes performances d'extraction répondant à la majorité des prédicats des requêtes, notamment les comparaisons d'égalité et d'intervalles.
- Répercussions efficaces des mises à jour, ne se dégradant pas en fonction d'une augmentation forte de la taille des tables.

Les principales opérations qu'un optimiseur réalise sur un index sont les suivantes :

- *index unique scan* passe par la racine de l'arbre, généralement toutes les colonnes de l'index sont concernées par une égalité dans le prédicat `WHERE`. Manière la plus optimale en principe, mais qui n'est pas toujours utilisée par l'optimiseur au profit de *range scan*.
- *index range scan* passe par la racine de l'arbre et accède séquentiellement aux blocs feuilles (doublement chaînés). Opération très utilisée par l'optimiseur, notamment lorsqu'une colonne de l'index est concernée par une inégalité dans le prédicat `WHERE` et que l'index n'est pas unique. Dans tous ces cas, l'optimiseur juge qu'il est plus rapide de parcourir les feuilles de l'index plutôt que l'index lui-même.

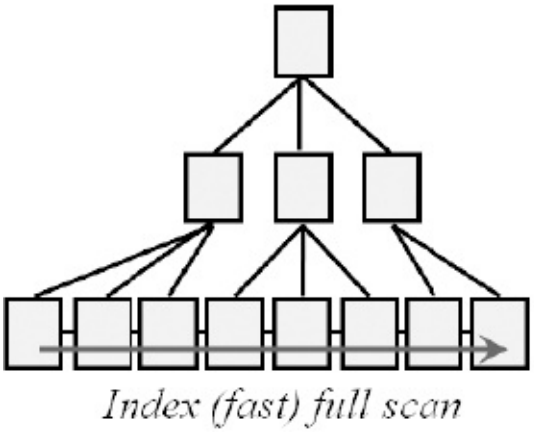
Figure 10-8 *Accès direct et parcours par intervalle d'un index B-tree*

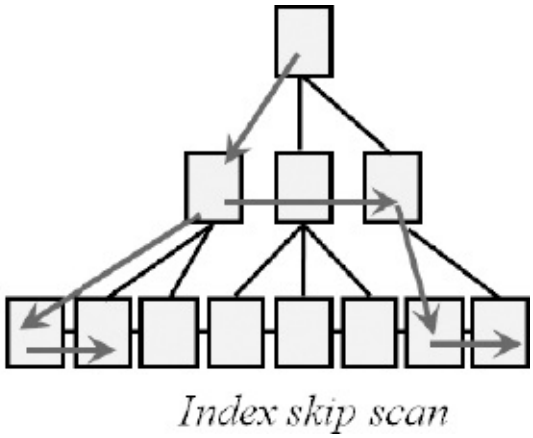


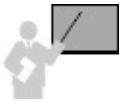


- *index full scan* est une alternative au parcours *full table scan* quand l'index contient toutes les colonnes nécessaires à la requête (index couvrant) et qu'au moins une de ces colonnes est `NOT NULL`.
- *index skip scan* (concerne les index multicolonnes) utilise l'index alors que la (les) première(s) colonne(s) de l'index n'est (ne sont) pas présente(s) dans le prédicat `WHERE`. Ce parcours d'index n'est pas implémenté avec MySQL.

Figure 10-9 *Parcours séquentiel et par saut d'un index B-tree*







Les tables InnoDB sont constituées en index *cluster* (ce qu'Oracle appelle *index organized tables*) basé sur les valeurs de la clé primaire (s'il n'existe aucune clé, MySQL en crée une sous la forme d'un entier de 6 octets). Dans un tel index, les feuilles ne référencent pas des adresses dans des blocs de données mais contiennent les enregistrements entiers de la table. L'index est la table et

inversement.

Il est possible de créer des index sur des colonnes BLOB ou TEXT pour les tables associées aux moteurs MyISAM et InnoDB.

Pour vous convaincre de l'utilité de l'indexation, exécutez la requête suivante sans index. Il s'agit d'une division qui extrait les adhérents inscrits exactement à tous les sports (à noter que vous auriez pu écrire cette requête sans division en comptant tous les sports, et comparer ce nombre au total des lignes de la table Pratique groupées sur le code adhérent. Nous procédons ainsi pour l'exemple).

```
SELECT a.civilite, a.prenom, a.nom, a.tel FROM Adherent a
      WHERE NOT EXISTS
        (SELECT spid FROM Sport
                    WHERE spid NOT IN
                   (SELECT spid FROM Pratique WHERE adhid = a.adhid))
      AND NOT EXISTS
        (SELECT spid FROM Pratique
                    WHERE adhid = a.adhid
                    AND spid NOT IN
                        (SELECT spid FROM Sport));
+----------+---------+----------+---------------+
| civilite | prenom  | nom      | tel           |
+----------+---------+----------+---------------+
| Mme.     | CELINE  | LARRAZET | 08-31-48-45-65 |
| Mlle.    | JENIFER | THIRIET  | 06-63-71-28-54 |
+----------+---------+----------+---------------+
2 rows in set (6 min 37.93 sec)
```

Pour découvrir que les deux super sportives sont Céline Larrazet et Jenifer Thiriet, vous allez devoir patienter plusieurs minutes sans index (dans mon cas près de 7 minutes). Cette même réponse vous sera fournie quasi instantanément si les index adéquats sont présents (surtout celui portant sur la colonne adhid jouant le rôle de clé étrangère dans la table Pratique). Cette différence est éloquente même pour une volumétrie relativement réduite (de l'ordre de 24 000 adhérents). Le tableau suivant présente la création des index (unique) sur les clés primaires et sur les clés étrangères (non uniques) des tables de cet exemple.

Tableau 10-13 Création d'index sur les clés

Index sur les clés primaires	Index sur clés étrangères

`ALTER TABLE Sport`
`  ADD CONSTRAINT pk_Sport`
`  PRIMARY KEY (spid);`
`ALTER TABLE Adherent`
`  ADD CONSTRAINT pk_Adherent`
	`CREATE INDEX idx_adhid_pratique USING BTREE`
	`  ON Pratique (adhid)`
	`  COMMENT 'numéro adhérent - clé étrangère';`
	`CREATE INDEX idx_spid_pratique USING BTREE`
	`  ON Pratique (spid)`

```
      PRIMARY KEY (adhid);                    COMMENT 'numéro sport - clé étrangère';
```

- Réduisez au maximum la taille de la (des) colonne(s) composant une clé primaire, vous réduirez ainsi la taille de l'index et les temps de parcours.
- Créez un index pour chaque clé étrangère afin de rendre plus efficaces les jointures.
- Vérifiez les index de chaque table avec `SHOW INDEX FROM` *`nomTable`*.

Si une colonne d'une table est supprimée (`ALTER TABLE … DROP [COLUMN] …`), la colonne est également supprimée des index où elle figure. Dans le cas d'un index monocolonne, l'index disparaît, alors qu'un index multicolonne se reconstruit sans la colonne supprimée.

Si une colonne d'un index est modifiée (option `CHANGE` ou `MODIFY` de la commande `ALTER TABLE`), le nouveau type de la colonne est répercuté dans les index associés.

La suppression d'un index s'opère par la commande `ALTER TABLE`. Le tableau suivant décrit la suppression des index sur les clés des tables de l'exemple.

Tableau 10-14 Supression des index sur les clés

Index sur les clés primaires	Index sur clés étrangères

`ALTER TABLE Sport`
`  DROP PRIMARY KEY;`
`ALTER TABLE Adherent`
`  DROP PRIMARY KEY;`
	`ALTER TABLE Pratique`
	`  DROP INDEX idx_adhid_pratique;`
	`ALTER TABLE Pratique`
	`  DROP INDEX idx_spid_pratique;`

Bien que les index *B-tree* de MySQL soient majoritairement employés, ils ne s'appliquent pas aux conditions suivantes :

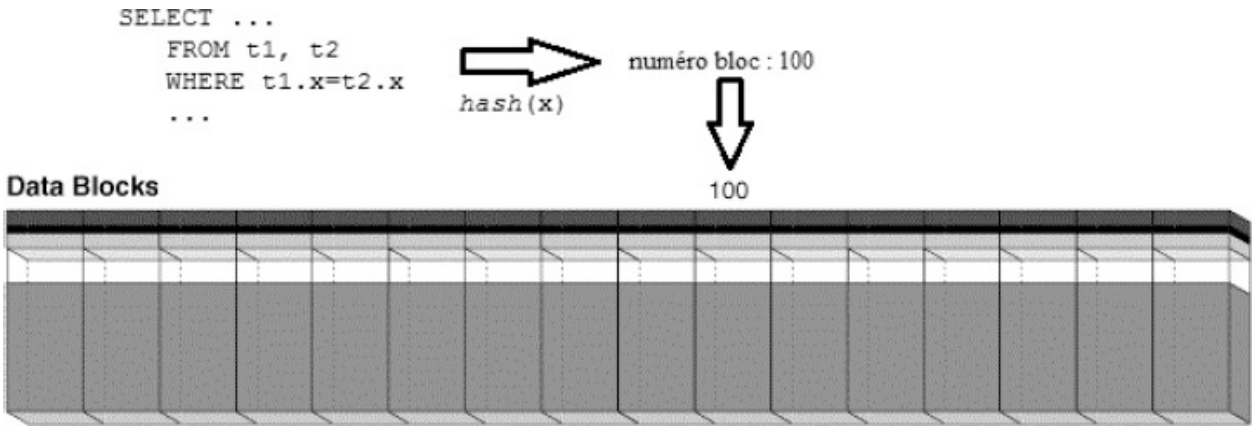
- Données de faible cardinalité (colonne de relativement peu de valeurs distinctes en regard du nombre total de lignes). Dans notre exemple, il ne servira à rien de créer un index sur la colonne `civilite` dont la cardinalité est réduite à `'Mlle.'`, `'Mme.'` ou `'Mr.'`. Un index *bitmap* serait une alternative mais cette technique n'est pas encore implémentée.

- Quand la condition de recherche comporte une fonction SQL appliquée à une colonne, aucun index ne peut être utilisé (par exemple, l'index sur la colonne nom sera inopérant à la suite d'une condition telle que `WHERE UPPER(nom)='DIFFIS'`). Comme il n'existe pas d'index basé sur les fonctions, vous devrez opérer vos fonctions SQL au niveau de la variable et non de la colonne testée : `WHERE nom=`*`fonction`*`('DIFFIS')`.

Index hash

Les index *hash* sont basés sur une table de hachage qui renvoie, pour chaque valeur de l'index, un numéro de bloc (qui contient la ou les lignes associées). La fonction de hachage permet, en théorie, d'associer un numéro différent à chaque valeur distincte de l'index. En conséquence, ce mécanisme d'accès aux données est le plus performant de tous (même par rapport aux *B-tree*) pour les recherches basées sur les égalités (par exemple, les prédicats `adhid=5` ou `adhid<>4`). Il ne s'apparente pas une recherche dichotomique mais signifie un accès direct.

Figure 10-10 *Parcours d'un index hash*



En revanche, ce mécanisme est pénalisant (même par rapport à un accès séquentiel) pour les recherches basées sur les inégalités (par exemple, les prédicats `adhid>5` ou `adhid BETWEEN 4 AND 60`), car les numéros de blocs ne sont pas triés en fonction des valeurs de l'index mais dépendent de la fonction de hachage (qui renverra peut-être 100 pour un index de valeur 4 et 2000 pour un index de valeur 5).

Bien que l'indexation en *hash* peut être utilisée en interne par MySQL pour éventuellement remplacer un accès en *B-tree* (on parle de l'algorihme *hash adaptative*), cette indexation ne s'applique pas encore explicitement aux tables `InnoDB` et `MyISAM`.

Seules les tables `MEMORY` peuvent être associées explicitement à des index *B-tree* et *hash* (ce dernier étant le type d'index par défaut).

La création d'un index *hash* pour les adhérents nécessite au préalable de transformer le moteur de stockage de cette table. Le tableau suivant détaille ces opérations.

Tableau 10-15 Création d'un index hash et B-tree

Commandes SQL	Commentaires
`ALTER TABLE Adherent **ENGINE = MEMORY**;`	Modification du moteur associé pour la table des adhérents.

`ALTER TABLE Adherent`
`  ADD CONSTRAINT pk_Adherent`
`  PRIMARY KEY (adhid);`
	Création de la clé primaire (index *hash* par défaut).

`CREATE INDEX idx_btree_adhid`
`  USING BTREE`
`  ON Adherent (adhid);`
	Création d'un index *B-tree* sur la colonne clé.

En présence de deux index uniques (un *B-tree* et un *hash*) sur la (les) même(s) colonne(s), on parle de « surindexation » qui peut induire un surcoût en mises à jour. Selon la volumétrie et les cardinalités, l'optimiseur optera pour l'un ou l'autre des index. Le tableau suivant présente pour quelques requêtes, l'index utilisé.

Tableau 10-16 Surindexation

Index *hash* utilisé	Index *B-tree* utilisé

`SELECT adhid, nom, tel, prenom`
`  FROM  Adherent`
`  WHERE adhid=5674;`
`SELECT adhid, nom, tel, prenom, civilite`
`  FROM  Adherent`
`  WHERE adhid IN`
`(5674,34,56,790,25999);`
	`SELECT adhid, nom, tel, prenom`
	`  FROM  Adherent`
	`  WHERE adhid>5674 AND adhid<17890;`
	`SELECT adhid, nom, tel, prenom`
	`  FROM  Adherent`
	`  WHERE adhid>5674;`

En présence de valeurs en doublon (par exemple, les clés étrangères) au niveau de la ou des colonnes indexées, ou de fréquentes recherches par inégalités, évitez de définir un index *hash* et préférez un index *B-tree*.

Index et NULL

Depuis la version 5.5, le mécanisme des index *B-tree* permet d'associer des entrées à des `NULL` (tables `MyISAM`, `InnoDB` et `MEMORY`). Il est toutefois souhaitable qu'un index ne concerne que des colonnes non nulles pour des raisons de performances. Bien que le processus de normalisation vise à éviter les `NULL`, cet objectif est un peu utopique.

Si vous manipulez des index sur des colonnes nulles, vous devez utiliser les prédicats `IS NULL` ou `IS NOT NULL` pour vos recherches, qui selon la cardinalité et la sélectivité de la colonne, feront intervenir ou non l'index.

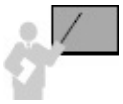
Le tableau suivant présente quelques résultats basés sur la recherche du nombre d'adhérents en fonction de leur numéro de téléphone (`NULL`, valeur, `NOT NULL`). 765 adhérents n'ont pas de téléphone (3 % de la population). L'index n'est pas utilisé pour extraire les adhérents ayant un numéro de téléphone (un *full scan table* est préféré pour balayer les 97 % des données).

Tableau 10-17 Index B-tree et colonne `NULL`

`SELECT COUNT(nom) FROM Adherent WHERE…`	**type**	**possible_keys**	**key**
Sans index			
`tel IS NULL`	ALL	ALL	NULL
`tel='06-81-94-44-31'`	ALL	ALL	NULL
`tel IS NOT NULL`	ALL	ALL	NULL
Index *B-tree :* `CREATE INDEX idx_tel_btree ON Adherent (tel);`			
`tel IS NULL`	ref	idx_tel_btree	idx_tel_btree
`tel='06-81-94-44-31'`	ref	idx_tel_btree	idx_tel_btree
`tel IS NOT NULL`	ALL	idx_tel_btree	NULL

Index multicolonne

Un index peut être composé de plusieurs colonnes : on parle alors d'index multicolonne. Un exemple d'un tel index a été présenté à la section « Les colonnes UNIQUE ». Il concernait le nom, le prénom et le numéro de téléphone des adhérents.



Il est possible de restreindre la taille (on parle de préfixe d'index) d'une colonne d'un index de type `CHAR`, `VARCHAR`, `BINARY` et `VARBINARY`.

Un index sur une colonne `BLOB` ou `TEXT` doit être limité en taille par un préfixe (qui concerne les données géographiques et textuelles).

Un index multicolonne offre l'avantage de pouvoir combiner des colonnes (ou expressions) présentant une faible sélectivité pour former un index dont la sélectivité est plus élevée. Par ailleurs, si toutes les colonnes concernées par une interrogation se trouvent dans un index composé, l'accès à l'index suffira (il n'est pas nécessaire d'accéder à la table).

Un index composé est utile principalement lorsque vos clauses `WHERE` font souvent référence à l'ensemble, ou à une partie des premières colonnes de l'index.

Choisissez la colonne de valeur la plus sélective afin de constituer la tête de l'index. Si plusieurs colonnes sont sélectives, considérez celle qui est le plus fréquemment utilisée. Dans certains cas, il vaut mieux utiliser un tel index que plusieurs index sur une colonne si les colonnes sont fréquemment testées simultanément.

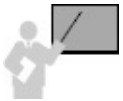
Le tableau suivant met en œuvre deux index : le premier est composé du nom, du prénom (restreint aux 18 premiers caractères) et du numéro de téléphone ; le second est composé du numéro de téléphone et du nom. L'optimiseur utilise naturellement le premier index pour répondre à la première requête. L'autre index est préféré pour la seconde requête car l'optimiseur juge la colonne `tel` plus sélective dans le prédicat.

Tableau 10-18 Index multicolonne

Création de l'index	Utilisé par la requête

`CREATE INDEX idx_nompretel_btree`
`  USING BTREE`
`  ON Adherent (nom, prenom(18), tel);`
	`SELECT adhid, civilite, nom, tel`
	`  FROM  Adherent`
	`  WHERE nom LIKE 'LARR%';`

`CREATE INDEX idx_telnom_btree`
`  USING BTREE`
`  ON Adherent (tel, nom);`
	`SELECT adhid, civilite, nom, tel`
	`  FROM  Adherent`
	`  WHERE nom LIKE 'LARR%'`
	`  AND  tel='08-31-48-45-65';`



Si vous évaluez des expressions de la forme `nom LIKE '%LARR%'` ou `tel LIKE '%08%'`, aucun index ne sera utilisé par l'optimiseur (un *full scan table* sera préféré).

Index textuels

MySQL 5.6 et ultérieures

Depuis la version 5.6, l'indexation textuelle (FULLTEXT) est possible avec les moteurs InnoDB et MyISAM et elle concerne des colonnes CHAR, VARCHAR ou TEXT. Avant la version 5.6, seules les tables de type MyISAM pouvaient en bénéficier.

Ce mécanisme fournit d'une part une alternative à la limite des index traditionnels pour répondre aux prédicats suivants : *colonne* `LIKE '%chaine%'`. D'autre part, il sera possible d'extraire seulement des enregistrements réellement pertinents (affectation d'un poids à chaque mot recherché).

Pour des raisons de performances, et lors d'une importation volumineuse de données, il est important de créer les index FULLTEXT après le chargement des données. En d'autres termes, pour définir un index textuel, préférez les commandes ALTER TABLE ou CREATE INDEX plutôt que CREATE TABLE.

La recherche sur un ou plusieurs mots-clés se programme comme un prédicat dans une requête par la fonction MATCH… AGAINST… La syntaxe est la suivante :

```
MATCH (colonne1, colonne2,...)
   AGAINST (expression  [ { IN BOOLEAN MODE
                         | IN NATURAL LANGUAGE MODE
                         | IN NATURAL LANGUAGE MODE WITH QUERY EXPANSION
```

```
                        | WITH QUERY EXPANSION  }])
```

Il existe trois types de recherches textuelles :

- La recherche booléenne (`IN BOOLEAN MODE`) permet de préciser des recherches sur plusieurs mots-clés (le signe `+` permet d'ajouter un mot-clé à la recherche, le signe `-` d'exclure un mot-clé, et un espace vide entre deux mots-clés indique un `OR`). La liste des mots interdits (*stopwords*) se trouve dans le ficher `ft_static.c` du répertoire `storage/myisam/` (vous devrez travailler avec la distribution source).
- La recherche naturelle (`IN NATURAL LANGUAGE MODE`) sans opérateurs. La liste des mots interdits s'applique. Les mots cherchés qui sont présents dans plus de la moitié du texte des colonnes ne sont pas considérés.
- La recherche étendue (`IN NATURAL LANGUAGE MODE WITH QUERY EXPANSION` ou `WITH QUERY EXPANSION`) consiste en une extension de la recherche précédente. Les mots associés à la ligne qui se rapproche le plus de l'expression cherchée sont ajoutés à l'expression au cours d'une seconde recherche pour laquelle les résultats finaux sont extraits.

Les paramètres système disponibles pour modifier le comportement par défaut du serveur sont listés dans le tableau suivant. La modification d'une variable (notamment dans les fichiers `myisam/ftdefs.h` et `myisam/fulltext.h`) nécessite de reconstruire les index textuels existants.

```
mysql> SHOW VARIABLES LIKE 'ft%';
+--------------------------+----------------+
| Variable_name            | Value          |
+--------------------------+----------------+
| ft_boolean_syntax        | + -><()~*:""&| |
| ft_max_word_len          | 84             |
| ft_min_word_len          | 4              |
| ft_query_expansion_limit | 20             |
| ft_stopword_file         | (built-in)     |
+--------------------------+----------------+
```

Tableau 10-19 Paramètres pour les index textuels

Paramètres	Significations
`ft_boolean_syntax`	Opérateurs autorisés pour une recherche en mode booléen.
`ft_max_word_len`	Taille minimale des mots indexés.
`ft_min_word_len`	Taille maximale des mots indexés.
`ft_query_expansion_limit`	Nombre maximal des mots à indexer lors

	d’expansions.
`ft_stopword_file`	Fichier contenant les mots à ne pas indexer (articles, prépositions, etc.). Une chaîne vide permet de ne pas utiliser de liste.

Considérons la table suivante et supposons qu’elle contient un certain nombre d’articles d’un forum consacré à MySQL. Les colonnes `titre` et `texte` contiennent respectivement le titre et le contenu de l’article.

Tableau 10-20 Création d’un index textuel

Création de la table	Création de l’index

`CREATE TABLE forums_MySQL`
`(entryID INT(9) NOT NULL AUTO_INCREMENT,`
`titre VARCHAR(80) NOT NULL,`
`texte TEXT NOT NULL,`
`PRIMARY KEY (entryID)) ENGINE=MyISAM;`
	`CREATE FULLTEXT INDEX idx_forums_MySQL`
	`ON forums_MySQL (titre,texte);`

Recherchons à présent les articles qui vérifient certaines conditions. Si le titre ou le texte ne contient pas les expressions recherchées, ou si tous les mots appartiennent à la liste des mots interdits, aucun résultat ne sera retourné. Il en va de même si les mots sont plus courts que le minimum autorisé (ici, 4 caractères) ou si plus de la moitié des lignes vérifient la condition (pour une recherche en langage naturel).

Tableau 10-21 Utilisation d’un index textuel

Requêtes	Commentaires
`mysql> SELECT entryID, titre` `FROM forums_MySQL` `WHERE MATCH (titre,texte) AGAINST ('SELECT' IN BOOLEAN MODE);` `+---------+----------` `----------+` `| entryID |` `titre |` `+---------+----------` `----------+` `| 1	ORDER BY [une certaine valeur en 1er] |` `| 2	gerer une erreur de script sur un DROP USER ou un REVOKE |` `| 3	Variables dans un trigger |` `| 4	Gestion d'exception dans une procédure stockée |` `| 6	requete imbriquée sur une meme table |`	Recherche des articles contenant l’expression `SELECT`.

```
|       7 | probleme syntaxe avec alias de
table                     |
+---------+-----------------------------------------------
----------+
```

```
mysql> SELECT entryID, titre
       FROM forums_MySQL
     WHERE MATCH (titre,texte)
           AGAINST ('SELECT' IN NATURAL LANGUAGE MODE);
Empty set (0.00 sec)
```

Plus de la moitié des articles contient l'expression SELECT.

```
mysql> SELECT entryID, titre, MATCH (titre,texte)
       AGAINST ('SELECT INSERT' IN NATURAL LANGUAGE
MODE) AS score
   FROM forums_MySQL
   WHERE MATCH (titre,texte)
         AGAINST ('SELECT INSERT' IN NATURAL LANGUAGE
MODE)
   ORDER BY score DESC;
+---------+---------------------------------------+------
-------+
| entryID | titre                                 |
score       |
+---------+---------------------------------------+------
-------+
|       8 | Insert ou Update probleme avec
…       |  0.52928211 |
|       3 | Variables dans un
trigger              |  0.28339432 |
|       4 | Gestion d'exception dans une
…         |  0.19842221 |
+---------+---------------------------------------+------
-------+
```

Recherche des articles contenant au moins une des expressions SELECT ou INSERT.

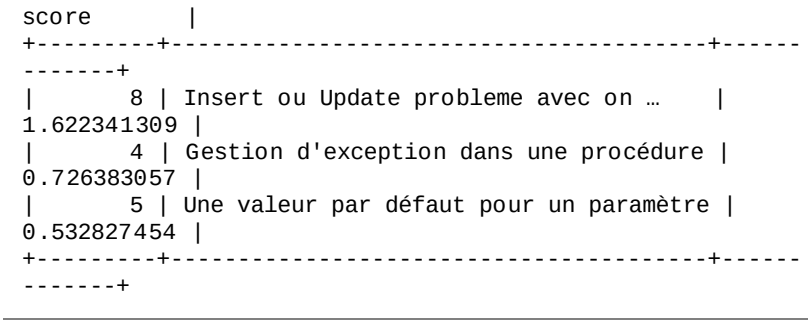
```
Même requête avec ('+SELECT +DELETE' IN NATURAL LANGUAGE
MODE)
+---------+---------------------------------------+------
-------+
| entryID | titre                                 |
score       |
+---------+---------------------------------------+------
-------+
|       2 | gerer une erreur de script sur …      |
1.083374834 |
+---------+---------------------------------------+------
-------+
```

Recherche des articles contenant les expressions SELECT et DELETE.

```
Même requête avec ('+SELECT PROCEDURE' IN NATURAL
LANGUAGE MODE)
+---------+---------------------------------------+------
-------+
| entryID | titre                                 |
score       |
+---------+---------------------------------------+------
-------+
|       4 | Gestion d'exception dans une procé…   |
0.725921357 |
|       5 | Une valeur par défaut pour un param…  |
0.532827454 |
+---------+---------------------------------------+------
-------+
```

Recherche des articles contenant l'expression SELECT et les ordonne de façon plus pertinente s'ils contiennent aussi l'expression PROCEDURE.

```
Même requête avec ('+UPDATE -PROCEDURE' IN NATURAL
LANGUAGE MODE)
+---------+---------------------------------------+------
-------+
| entryID | titre                                 |
```

	Recherche des articles contenant l'expression UPDATE mais pas l'expression PROCEDURE.
Même requête avec ('**+SELECT +(>UPDATE <PROCEDURE)**' IN NATURAL LANGUAGE MODE)	Recherche des articles contenant à la fois les expressions SELECT et UPDATE ou SELECT et PROCEDURE mais en considérant les derniers comme moins pertinents.
Même requête avec ('**"CREATE T*"**' IN NATURAL LANGUAGE MODE)	Recherche des articles contenant une expression CREATE TABLE, CREATE TRIGGER, etc. Les doubles guillemets permettent de rechercher une expression composée de plusieurs mots.

Reconstruction des index

Selon la fréquence de mise à jour et le volume de données modifiées, il est important de reconstruire les index *B-tree* du fait de leur fragmentation. Ceci est particulièrement notable quand les colonnes index sont de taille variable (VARCHAR, VARBINARY, BLOB ou TEXT). Plus la densité des blocs feuilles est élevée, meilleur est l'index. À l'inverse, il est souhaitable de reconstruire l'index lorsqu'il contient de nombreux blocs dont la cardinalité est faible. La reconstruction d'un index existant offre, dans certains cas, de meilleures performances que la destruction de celui-ci, puis sa recréation.

La commande OPTIMIZE TABLE (disponible pour les tables MyISAM, InnoDB et ARCHIVE) effectue un certain nombre de tâches : la collecte des statistiques, la reconstruction des index, la récupération d'espace inutilisé et la

défragmentation. MySQL verrouille la table durant la phase d'optimisation. Les privilèges `SELECT` et `INSERT` sont requis pour chaque table analysée.

Réorganisons la table des adhérents (`InnoDB`) et celle des articles (`MyISAM`) :

Tableau 10-22 Réorganisation de deux tables

Commande	Résultat

`mysql> OPTIMIZE NO_WRITE_TO_BINLOG TABLE Adherent, forums_mysql;`	`+----------------------+----------+----------+----------+`
	`| Table | Op | Msg_type | Msg_text |`
	`+----------------------+----------+----------+----------+`
	`| bdsoutou.adherent | optimize | note | Table does not support |`
	`| optimize, doing recreate + analyze instead |`
	`| bdsoutou.adherent | optimize | status | OK |`
	`| bdsoutou.forums_mysql | optimize | status | OK |`
	`+----------------------+----------+----------+----------+`

Choix d'un index par l'optimiseur

Plusieurs autres facteurs influencent l'optimiseur sur le choix d'utiliser un index ou non :

- Le volume des tables comportant relativement peu de lignes vérifiant une condition ne nécessite pas un parcours indexé. Au contraire, si de nombreux blocs sont concernés, le parcours de la table sera choisi.
- Une directive (*hint*) incompatible comme `IGNORE INDEX (…)`.
- Le partitionnement de la table et l'absence d'index global.
- L'utilisation de variables de session (*bind variables*). L'optimiseur établit un plan d'exécution sans connaître a priori la valeur de ces variables.

Contrôle d'index et d'opérateurs

Depuis la version 5.6, le mécanisme ICP (*Index Condition Pushdown*) permet des optimisations de requêtes qui utilisent des index sans explorer aucune table (lorsque vous êtes en présence du mot-clé `Index only` dans un plan d'exécution). Disponible pour les moteurs InnoDB et MyISAM, depuis la version 5.7, ce procédé peut s'appliquer aux partitions. Pour visualiser les paramètres de votre

session :

```
mysql>  show variables like 'optimizer_switch';
+------------------+-------------------------------------------------+
| Variable_name    | Value                                           |
+------------------+-------------------------------------------------+
| optimizer_switch |
index_merge=on,index_merge_union=on,index_merge_sort_union=on,index_merge_
intersection=on,engine_condition_pushdown=on,index_condition_pus-
hdown=on,mrr=on,mrr_cost_based=on,block_nested_loop=on,batched_key_
access=off,materialization=on,semijoin=on,loosescan=on,first-
match=on,subquery_materialization_cost_based=on,use_index_extensions=on,condition_fanout_
filter=on,derived_merge=on    |
+------------------+-------------------------------------------------+
```

Il est possible de modifier ces paramètres au niveau de la session ou du serveur. Pour ce faire, utilisez une des quatre options au niveau de chaque commande (`default` pour revenir à tous les paramètres par défaut, `nom_option=default` pour positionner une option à sa valeur par défaut, `nom_option=off|on` pour désactiver ou activer l'option).

```
SET [GLOBAL|SESSION] optimizer_switch='commande[,commande]...';
```

Jointures

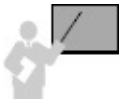
Les jointures ont été étudiées au chapitre 4. Il existe ainsi plusieurs écritures pour programmer une interrogation mettant en relation plusieurs tables. La majorité des jointures est basée sur l'égalité entre une clé étrangère et une clé primaire (ou unique). Dans toute jointure entre deux tables, une ligne d'une table est appelée *inner*, l'autre *outer*.

Pour choisir un plan d'exécution, l'optimiseur décide de la stratégie en fonction de plusieurs facteurs :

- Le chemin d'accès afin d'extraire les données de chaque table lors de la jointure.
- La méthode de jointure pour chaque paire de lignes jointes, l'algorithme adopté par MySQL est la boucle imbriquée (*nested loop*). Il existe d'autres algorithmes qui ne sont pas mis en œuvre par MySQL, à savoir les algorithmes *sort merge join*, *cartesian join* et *hash join*.
- L'ordre dans lequel les jointures doivent se réaliser lorsqu'il y a plus de deux tables en relation. La deuxième jointure s'opère après la première, etc.

L'optimiseur détermine d'abord si la jointure retourne au final au moins une ligne. Cette réponse est basée sur les contraintes `UNIQUE` et `PRIMARY KEY` des tables. Si ces contraintes existent, l'optimiseur traite ces tables en premier, puis rend optimale la suite des opérations en minimisant les coûts (via les statistiques) des *nested loops* en se basant sur le coût des lectures en mémoire de chaque ligne de la table *outer* et chaque correspondance avec les lignes de la table *inner*.

Nested loops



Une opération *nested loop* se déroule en trois temps. L'optimiseur choisit tout d'abord la table qui « pilote » l'itération (*outer table*), puis désigne l'autre table en tant que *inner*. Ensuite, pour chaque ligne de la table *outer*, toutes les lignes associées de la table *inner* sont extraites.

Les jointures programmées avec l'opérateur *nested loop* sont très performantes lorsqu'un faible nombre de données de la première table (*outer*) est mis en jointure et que la condition de jointure accède efficacement à la deuxième table (*inner*).

L'exemple suivant décrit une équijointure entre trois tables (ici, on extrait l'identité des adhérents inscrits à l'escrime et ceux inscrits au ping-pong). Ces deux écritures produisent en général le même plan d'exécution.

Tableau 10-23 Jointure mise en œuvre avec nested loop

Jointure relationnelle	Jointure SQL2

`SELECT a.adhid, a.nom, a.tel`
`FROM Adherent a, Sport s, Pratique p`
`WHERE s.splibelle IN ('Escrime','Ping-pong')`
`AND a.adhid = p.adhid`
`AND s.spid = p.spid;`
	`SELECT a.adhid, a.nom, a.tel FROM Adherent a`
	`INNER JOIN Pratique p ON p.adhid = a.adhid`
	`INNER JOIN Sport s ON s.spid = p.spid`
	`WHERE s.splibelle IN ('Escrime','Ping-pong');`

Le tableau suivant présente deux plans d'exécution possibles. Dans les deux plans, l'index de la table d'association est capital. Le premier plan décrit le parcours entier (*full scan table*) des adhérents, et pour chaque adhérent, l'index facilite la jointure avec la table `Sport`. Le second plan décrit le parcours entier

(*full scan table*) des sports pour restreindre ensuite la jointure avec les adhérents. C'est ce plan qui est le plus performant et qui sera sélectionné après la collecte des statistiques.

Tableau 10-24 Plans d'exécution d'une jointure

```
+-------------+-------+--------+-------------------------------------+------------------
-+---------+
| select_type | table | type   | possible_keys                       |
key                | key_len |
  ref              | rows  |
Extra                                                  |
+-------------+-------+--------+-------------------------------------+------------------
-+---------+
| SIMPLE      | a     | ALL    | PRIMARY                              |
NULL               | NULL    |
  NULL             | 24488
|                                                                      |


| SIMPLE      | p     | ref    | idx_adhid_pratique, idx_spid_pratique|
idx_adhid_pratique | 2       |
  bdsoutou.a.adhid |     1
|                                                                      |


| SIMPLE      | s     | eq_ref | PRIMARY                             |
PRIMARY            | 1       |
  bdsoutou.p.spid  |     1 | Using
where                                                          |
+-------------------------------------------------------------------------------------
-----------+
```

```
+-------------+-------+--------+-------------------------------------+------------------
-+---------+
| select_type | table | type   | possible_keys                       |
key                | key_len |
  ref              | rows |
Extra                                                    |
+-------------+-------+--------+-------------------------------------+------------------
-+---------+
| SIMPLE      | s     | ALL    | PRIMARY                              |
NULL               | NULL    |
  NULL             |   12 | Using
where                                                        |


| SIMPLE      | p     | ref    | idx_adhid_pratique,idx_spid_pratique |
idx_spid_pratique  | 1       |
  bdsoutou.s.spid  | 1587
|                                                                        |


| SIMPLE      | a     | eq_ref | PRIMARY                             |
PRIMARY            | 2       |
  bdsoutou.p.adhid |    1
|                                                                        |
+-------------------------------------------------------------------------------------
-----------+
```

Les sous-interrogations sont à éviter pour programmer des jointures, car selon la version de l'optimiseur, le plan d'exécution peut se révéler être moins performant que celui d'une jointure relationnelle ou ANSI.

Comme l'illustrent la requête et le plan suivants, la première itération met en jeu la table des adhérents (*outer* qui ne filtre rien) qui est combinée à la table d'association (*inner*) qui sera accédée par un index. Pour chaque ligne extraite de cette jointure, l'accès à la table des sports est indexé.

Tableau 10-25 Jointure procédurale et plan d'exécution

```
SELECT adhid, nom, tel
    FROM Adherent WHERE adhid IN
            (SELECT adhid
                 FROM Pratique WHERE spid IN
                        (SELECT spid FROM Sport WHERE splibelle IN ('Escrime','Ping-
pong')));
```

```
+----+--------------------+----------+----------------+--------------------+-------------
------+---------+
| id | select_type        | table    | type           | possible_keys      |
key               |key_len  |
  ref  | rows  |
Extra                                                                                 |
+----+--------------------+----------+----------------+--------------------+-------------
------+---------+
|  1 | PRIMARY            | Adherent | ALL            | NULL               |
NULL              | NULL    |
 NULL | 24238 | Using
where                                                                                 |


|  2 | DEPENDENT SUBQUERY | Pratique | index_subquery  | idx_adhid_pratique |
idx_adhid_pratique |  2      |
 func |     1 | Using
where                                                                                 |


|  3 | DEPENDENT SUBQUERY | Sport    | unique_subquery | PRIMARY            |
PRIMARY           |  1      |
 func |     1 | Using
where                                                                                 |
+----+--------------------+----------+----------------+--------------------+-------------
------+---------+
```

Les performances de cette jointure sont très mauvaises car il n'est pas nécessaire d'examiner toutes les inscriptions pour tester le sport pratiqué. Sans

index, cela devient catastrophique. Testez les trois écritures de ces jointures et vous constaterez par vous-même qu'il vaut mieux éviter les sous-requêtes dans la mesure du possible.

Index invisibles

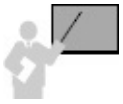
Les index invisibles masquent à l'optimiseur leur existence avant l'exécution d'une requête et permettent de tester différents scénarios d'optimisation. Rendre invisible un index évite de le détruire pour éventuellement plus tard le recrééer. En outre, un index invisible n'affecte pas la maintenance car il continue de se mettre à jour après toute modification des colonnes indexées.

Tableau 10-26 Création d'index invisibles

Instruction	Commentaire

`CREATE TABLE bdutil.clients`
`(id_cli SMALLINT AUTO_INCREMENT PRIMARY KEY,`
`code_cli VARCHAR(3), nom_cli VARCHAR(30),`
`mail_cli VARCHAR(60), date_nais DATE ,`
`INDEX idx_code_cli (code_cli) INVISIBLE`
`) DEFAULT CHARACTER SET latin1`
`COLLATE latin1_bin;`
	L'index invisible est créé en même temps que la table.

`CREATE INDEX idx_nom_cli` `ON bdutil.clients(nom_cli) INVISIBLE;`	L'index invisible est créé après la table.
`ALTER TABLE bdutil.clients` `ADD INDEX idx_date_nais (date_nais) INVISIBLE;`	



L'index sur la clé primaire ne peut pas être rendu invisible. Il en va de même pour un unique index d'une table non pourvue de clé primaire.

L'information de visibilité sera disponible au niveau du dictionnaire des données (la base `INFORMATION_SCHEMA`) dans la vue `STATISTICS` à l'aide de la colonne `IS_VISIBLE` (`YES` ou `NO`).

La commande ALTER TABLE permet aussi de rendre visible à nouveau un index masqué. Ainsi, l'index portant sur le code du client est rendu visible comme suit.

```
ALTER TABLE bdutil.clients ALTER INDEX idx_code_cli VISIBLE;
```

Une fois les index déclarés visibles ou invisibles à la demande, le plan d'exécution ne prendra en compte que les index visibles.

Tableau 10-27 Plans d'exécution

Instruction	Commentaire
mysql>EXPLAIN SELECT id_cli, nom_cli, date_nais FROM bdutil.clients WHERE **code_cli LIKE 'RE%';**	+----+-------------+---------+-------+-------+------ | id | select_type | table | type | possible_keys +----+-------------+---------+-------+------------- | 1 | SIMPLE | clients | range | **idx_code_cli**
mysql> EXPLAIN SELECT id_cli, nom_cli, date_nais FROM bdutil.clients WHERE **nom_cli LIKE 'RE%';**	+----+-------------+---------+------+------------- | id | select_type | table | type | possible_keys +----+-------------+---------+------+------------- | 1 | SIMPLE | clients | ALL | **NULL** +----+-------------+---------+------+-------------

Configuration de l'optimiseur [les hints]

Une directive (*hint*) se place dans une requête après le nom de la table et impose à l'optimiseur d'utiliser ou non des index. La syntaxe est la suivante, elle fait intervenir des noms d'index et non des noms de colonnes (PRIMARY peut désigner l'index clé primaire).

```
SELECT … FROM nom_table [[AS] alias]
[  USE {INDEX|KEY}
      [{FOR {JOIN|ORDER BY|GROUP BY}] ([index1 [, index2…]])
 | IGNORE {INDEX|KEY}
      [{FOR {JOIN|ORDER BY|GROUP BY}] ([index1 [, index2…]])
 | FORCE {INDEX|KEY}
      [{FOR {JOIN|ORDER BY|GROUP BY}] ([index1 [, index2…]])
,…] …
```

- USE INDEX impose l'utilisation d'un ou plusieurs index. Si la liste qui suit est vide, aucun index ne sera utilisé.
- IGNORE INDEX inhibe un ou plusieurs index.
- FORCE INDEX agit comme USE INDEX tout en renseignant l'optimiseur sur le coût très important d'un *full scan* sur la table en question.

Le tableau suivant présente l'utilisation de ces *hints*. Les deux premières

requêtes forcent à effectuer un balayage entier de la table. La troisième requête oblige l'utilisation de l'index associé à la clé primaire. La dernière requête entraîne l'utilisation d'un index en particulier (s'il en existait plusieurs).

Tableau 10-28 Requêtes avec hint

Requête	Opération choisie
`SELECT nom, tel FROM Adherent IGNORE INDEX (PRIMARY) WHERE adhid = 20045;` `SELECT nom, tel FROM Adherent USE INDEX() WHERE adhid = 20045;`	Parcours entier de la table (*full scan*).
`SELECT nom, tel FROM Adherent FORCE INDEX (PRIMARY) WHERE adhid = 20045;`	Index clé primaire.
`SELECT adhid, nom, tel FROM Adherent USE INDEX (idx_tel3_btree) WHERE tel LIKE '05-%';`	

Depuis la version 5.7, il existe une autre façon d'écrire un *hint* au sein des instructions `EXPLAIN`, `SELECT`, `INSERT`, `REPLACE`, `UPDATE` et `DELETE` qui de plus sera prioritaire par rapport aux réglages à l'aide du paramètre optimizer_switch. Un *hint* peut s'appliquer au niveau global (affecte l'instruction entière), bloc (bloc à l'intérieur d'une instruction), table (affecte une table à l'intérieur d'un bloc), et au niveau des index. Consultez dans la documentation le chapitre 8.9.3 *Optimizer Hints* pour retrouver les paramètres de chaque *hint*.

Tableau 10-29 Résumé des hints internes

Hint	Affecte	Étendue
`BKA, NO_BKA`	Jointures *Batched Key*	bloc, table
`BNL, NO_BNL`	Jointures *Nested-Loop*	bloc, table
`MAX_EXECUTION_TIME`	Temps exécution	global
`MRR, NO_MRR`	Lectures *Multi-Range*	table, index
`NO_ICP`	Index Condition Pushdown	table, index
`NO_RANGE_OPTIMIZATION`	Optimisation *range*	table, index
`QB_NAME`	Nommer des blocs de requêtes	bloc
`SEMIJOIN, NO_SEMIJOIN`	Stratégies *semi-join*	bloc

`SUBQUERY`	Matérialisation de sous-requêtes	bloc

Comme Oracle le propose depuis longtemps, chacun de ces *hints* doit se trouver au sein d'une structure qui ressemble un peu à celle d'un commentaire. Plusieurs *hints* peuvent être utilisés simultanément dans la structure.

Tableau 10-30 Résumé des hints internes

Contexte	Exemple de syntaxe

Instruction LMD	`SELECT /*+ ... */ ...`
	`INSERT /*+ ... */ ...`
	`REPLACE /*+ ... */ ...`
	`UPDATE /*+ ... */ ...`
	`DELETE /*+ ... */ …`

Dans un bloc de requête	`(SELECT /*+ ... */ ...)`
	`(SELECT ...) UNION (SELECT /*+ ... */ ...)`
	`(SELECT /*+ ... */ ...) UNION (SELECT /*+ ... */ ...)`
	`UPDATE ... WHERE x IN (SELECT /*+ ... */ ...)`
	`INSERT ... SELECT /*+ ... */ ...`

Dans un plan d'exécution	`EXPLAIN SELECT /*+ ... */ ...` `EXPLAIN UPDATE ... WHERE x IN (SELECT /*+ ... */ ...)`

Vous devez utiliser un *hint* avec parcimonie, et uniquement après avoir collecté les statistiques et évalué le plan de l'optimiseur initial. Des modifications apportées à la base (structurelles et au niveau des données) peuvent rendre moins pertinentes, voire non valides, ce genre de directives.

Tables temporaires

Les vues (dématérialisées) étudiées au chapitre 5 permettent de simplifier l'écriture de requêtes particulièrement complexes mais n'apportent aucune garantie en ce qui concerne les performances. Dans le pire des cas, une vue peut être consommatrice de ressources si d'autres vues sont impliquées en cascade dans la requête.

Le principe d'une vue matérialisée (*materialized view*) est de fournir, à partir d'une requête, un résultat stocké (comme les lignes d'une table) et de le réactualiser à intervalles satisfaisants. Bien que MySQL ne propose pas ce

mécanisme, l'utilisation de tables temporaires peut répondre, en partie, à certaines problématiques de performance. Une table temporaire peut, par exemple, améliorer une jointure ou une sous-requête par le seul fait de stocker des lignes précalculées. De plus, l'indexation est possible.

Le seul problème concerne le rafraîchissement des données de la table temporaire (si des mises à jour sont opérées sur les tables initiales). Aucun procédé n'est automatisé et vous devrez programmer ces mises à jour, soit par déclencheur, soit par événement en recréant la table périodiquement. Bien que ces procédés s'avèrent coûteux si les volumes de données manipulés sont importants, vos requêtes seront néanmoins bien plus performantes.

À titre d'exemple, considérons la requête suivante, qui extrait les adhérents pratiquant exactement les mêmes sports que l'adhérente Céline Larrazet. Il s'agit d'une division (voir le chapitre 4) qui compare, pour chaque adhérent, les sports pratiqués avec les sports de l'adhérente en question (ensemble de référence).

```
SELECT a.civilite, a.prenom, a.nom, a.tel FROM Adherent a
   WHERE NOT EXISTS
    (SELECT spid FROM Pratique WHERE adhid IN
           (SELECT adhid FROM Adherent
                  WHERE nom = 'LARRAZET' AND prenom = 'CELINE')
        AND spid NOT IN (SELECT spid FROM Pratique
                         WHERE adhid = a.adhid))
    AND NOT EXISTS
       (SELECT spid FROM Pratique WHERE adhid = a.adhid
        AND spid
        NOT IN (SELECT spid FROM Pratique
                WHERE adhid IN
                (SELECT adhid FROM Adherent
                 WHERE nom = 'LARRAZET' AND prenom = 'CELINE'))
       )
    AND a.nom <> 'LARRAZET' AND a.prenom <> 'CELINE';
```

L'éxécution de cette requête avec les paramètres par défaut de MySQL prendra un certain temps (plus de 24 minutes dans ma configuration). Le plan d'exécution fait apparaître 7 requêtes dont 6 dépendantes (DEPENDANT SUBQUERY).

Utilisons une table temporaire pour définir cet ensemble de références (les sports de Céline Larrazet) qui est analysé deux fois dans la requête. En fait, vous devrez créer deux tables temporaires identiques car une table temporaire

ne peut être utilisée qu'une seule fois dans une requête.

Tableau 10-31 Création d'une table partitionnée en intervalle

Tables temporaires	Requête réécrite

`CREATE TEMPORARY TABLE temp1`
`AS SELECT spid FROM Pratique`
`WHERE adhid IN`
`(SELECT adhid FROM Adherent`
`WHERE nom = 'LARRAZET'`
`AND prenom = 'CELINE');`
`CREATE TEMPORARY TABLE temp2`
`AS SELECT * FROM temp1;`
	`SELECT a.civilite, a.prenom, a.nom, a.tel`
	`FROM Adherent a`
	`WHERE NOT EXISTS`
	`(SELECT spid FROM temp1 WHERE spid NOT IN`
	`(SELECT spid FROM Pratique`
	`WHERE adhid = a.adhid))`
	`AND NOT EXISTS`
	`(SELECT spid FROM Pratique`
	`WHERE adhid = a.adhid`
	`AND spid NOT IN`
	`(SELECT spid FROM temp2))`
	`AND a.nom <> 'LARRAZET'`
	`AND a.prenom <> 'CELINE';`

```
+----------+---------+---------+----------------+
| civilite | prenom  | nom     | tel            |
+----------+---------+---------+----------------+
| Mlle.    | JENIFER | THIRIET | 06-63-71-28-54 |
+----------+---------+---------+----------------+
1 row in set (0.58 sec)
```

Le plan d'exécution ne fait apparaître plus que 5 requêtes dont 4 sous-requêtes dépendantes. La requête s'exécute d'une manière bien plus efficace (moins d'une seconde pour savoir que seule Jenifer Thiriet pratique les sports en question).

Cet exemple est un peu caricatural, car il est préférable de s'intéresser aux index plutôt qu'à ajouter des redondances que ce soit au niveau de colonnes (on parlera de dénormalisation) ou au niveau des tables. Ici, l'ajout de deux index peut résoudre le problème (l'un portera sur le numéro des adhérents dans la table des inscriptions, l'autre sur les nom, prénom et téléphone des adhérents).

Partitionnement

Apparu en version 5.1 et en constante évolution, le partitionnement permet de décomposer une table volumineuse (et ses index) en parties de taille plus réduite : les partitions. Chaque partition est un objet nommé, composé de la

même structure qu'une table (colonnes et contraintes) et qui pourra disposer (dans le futur) de ses propres caractéristiques de stockage.

L'accès aux partitions est transparent (votre code n'aura pas à être réécrit si vous partitionnez une table). La stratégie de partitionnement pour une table comme pour un index présente de nombreux autres avantages :

- réduction des contentions sur des ressources partagées (bases OLTP) et amélioration des requêtes sur des *data warehouses* (bases OLAP).
- simplification de la maintenance (l'ajout, la suppression, la fusion ou la modification d'une partition sont possibles). Ces opérations évitent d'accéder à toute la table.
- optimisation des accès pour les extractions qui ne concernent qu'un nombre limité de partitions (on parle du mécanisme de *pruning*).

Vous pouvez adopter le partitionnement lors de la création de la table (`CREATE TABLE... PARTITION...`) ou modifier une table existante (`ALTER TABLE... PARTITION...`).

Pour savoir si vous pouvez bénéficier du partitionnement, et si vous utilisez un serveur MySQL d'une version antérieure à 5.6, interrogez la variable d'environnement suivante :

```
mysql> SHOW VARIABLES LIKE '%partition%';
+-------------------+-------+
| Variable_name     | Value |
+-------------------+-------+
| have_partitioning | YES   |
+-------------------+-------+
```

Pour les serveurs d'une version 5.6 et postérieures, listez les modules d'extension pour constater la présence de l'entrée suivante :

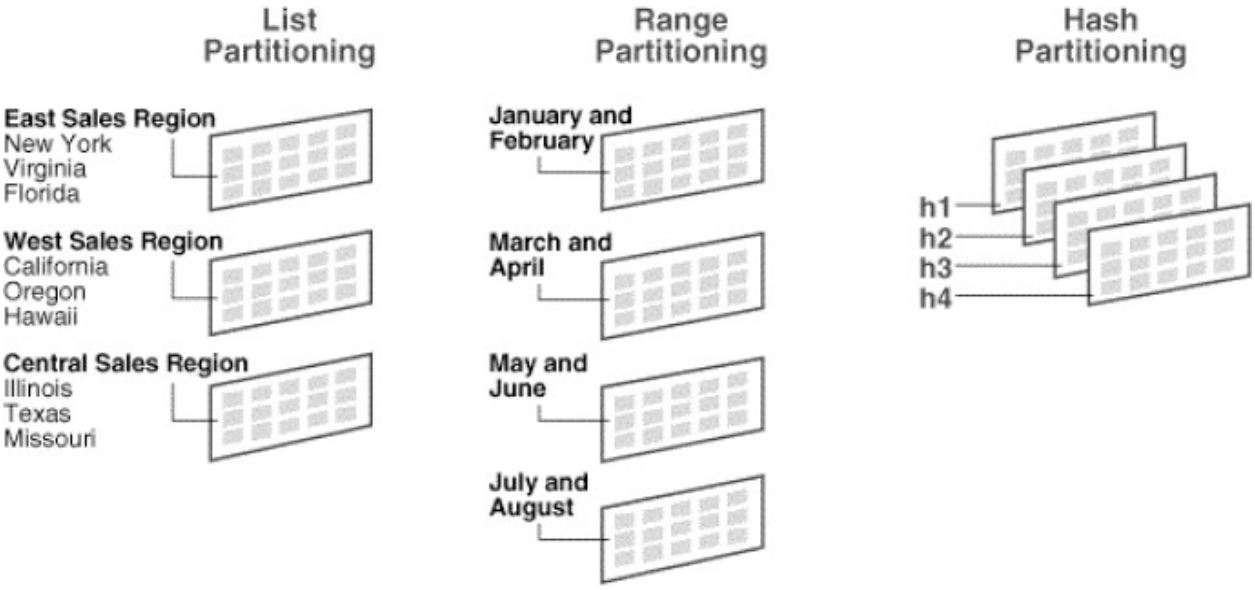
```
mysql> SHOW PLUGINS;
+------------+----------+----------------+---------+---------+
| Name       | Status   | Type           | Library | License |
+------------+----------+----------------+---------+---------+
| ...        | ...      | ...            | ...     | ...     |
| partition  | ACTIVE   | STORAGE ENGINE | NULL    | GPL     |
```

La clé de partition

La clé de partition est composée d'une ou de plusieurs colonnes qui déterminent la partition d'accueil de la ligne en question. Chaque ligne est affectée à une seule partition.

Les stratégies basiques de partitionnement (*single level partitioning*) sont l'intervalle (*range partitioning*), le hachage (*hash partitioning*) et la liste de valeurs (*list partitioning*). La figure suivante illustre ces types de partitions. La liste répartit les données selon les noms des régions, l'intervalle répartit les données de manière temporelle (ici, tous les deux mois) et le hachage répartit les données d'une manière homogène en utilisant un algorithme interne.

Figure 10-11 *Stratégies basiques de partitionnement © Document Oracle*



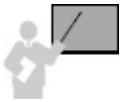
En principe, les tables devant être partitionnées :

- sont de taille importante (plus de 1 Go) ;
- contiennent des données historiques pour lesquelles des données récentes s'ajoutent à une nouvelle partition (par exemple, une table contenant des données modifiables sur le mois en cours et dans laquelle les informations concernant les mois précédents sont en lecture seule) ;
- sont celles qui nécessitent différentes unités de stockage.

Le partitionnement ne peut pas s'appliquer à une table MERGE, CSV ou FEDERATED.

Appliquons à présent ces différentes techniques de partitionnement dans des cas où il s'agit d'améliorer les performances de certaines requêtes.

Partitions par intervalle



Le partionnement *range* permet de répartir les lignes d'une table en fonction d'intervalles de valeurs de la clé de partitionnement. La directive `VALUES LESS THAN` indique la limite supérieure (sans l'inclure) de chaque partition. Toute ligne dont la colonne clé de la partition est égale ou supérieure à cette limite sera disposée automatiquement dans une des partitions suivantes. Toutes les partitions, à l'exception de la première, disposent implicitement d'une limite basse (limite haute de la partition qui précède) :

- Le mot-clé `MAXVALUE` est utilisé dans la dernière partition.
- L'expression `RANGE COLUMNS`, apparue à la version 5.5, permet d'enrichir le type de la clé (ici, une date).
- Toute ligne dont la clé de partitionnement est `NULL` est dirigée dans la première partition (celle qui définit la plus petite valeur).

La directive `PARTITION BY RANGE` précise les colonnes de partition et définit les intervalles.

En se basant sur la table `Adherent` qui contient des adhérents nés entre le 01/01/1920 et le 31/12/2005, comparons les deux implémentations (avec et sans partition). La table partitionnée par intervalles `Adherent_partition_range` est créée de la manière suivante : trois partitions sont définies. La première partition (`retraites`) contiendra les adhérents nés avant le 1er janvier 1945, la deuxième (`actifs`) contiendra les adhérents nés entre le 1er janvier 1945 et le 1er janvier 1993 et la dernière partition (`mineurs`) sera peuplée des adhérents nés après le 1er janvier 1993. Un index (local) est créé sur chaque partition.

Tableau 10-32 Création d'une table partitionnée par intervalle

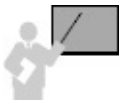
Code SQL	Commentaires
`CREATE TABLE Adherent_partition_range` `(adhid SMALLINT UNSIGNED NOT NULL,`	

```
  nom        VARCHAR(25) NOT NULL,                    Colonnes de la table.
  prenom     VARCHAR(30) NOT NULL,
  civilite   VARCHAR(12) NOT NULL,
  date_nais  DATE NOT NULL, tel VARCHAR(15))
PARTITION BY RANGE COLUMNS (date_nais)                 Déclaration de la clé de
 (PARTITION retraites VALUES LESS THAN ('1945-01-      partition.
01'),                                                  Définition des partitions.
  PARTITION actifs    VALUES LESS THAN ('1993-01-
01'),
  PARTITION mineurs   VALUES LESS THAN (MAXVALUE));
INSERT INTO Adherent_partition_range                   Chargement des données dans les
      SELECT * FROM Adherent;                          partitions.
CREATE INDEX Adherent_partition_idx                    Création d'un index local.
      ON Adherent_partition_range(date_nais);
```

Afin de comparer les deux implémentations, considérons la requête qui extrait l'identité des adhérents nés entre 1920 et 1945. Le plan d'exécution suivant montre que seule une partition sera analysée sans indexation car un grand nombre de lignes doit être extrait.

```
mysql> EXPLAIN PARTITIONS
     SELECT a.adhid, a.prenom, a.nom
       FROM Adherent_partition_range a
       WHERE a.date_nais >= '1920-01-01'
       AND   a.date_nais <  '1945-01-01';
+----+-------------+-------+------------+------+-----------------------+------+
---------+------+-------+-------------+
| id | select_type | table | partitions | type | possible_keys         | key  |
 key_len | ref  | rows  | Extra       |
+----+-------------+-------+------------+------+-----------------------+------+
|  1 | SIMPLE      | a     | retraites  | ALL  | Adherent_partition_idx | NULL |
 NULL    | NULL | 24561 | Using where |
+----+-------------+-------+------------+------+-----------------------+------+
```

Partitions par liste



Le partitionnement par liste permet de répartir les lignes d'une table en fonction d'un ensemble fini de valeurs de la clé de partitionnement. L'avantage de ce mécanisme est qu'il permet de faire intervenir une sémantique dans la clé de partionnement (pas de notion d'ordre comme pour les intervalles).

L'expression `LIST COLUMNS`, apparue à la version 5.5, permet d'enrichir le type de la clé (ici, une chaîne de caractères).

Depuis MySQL 5.6, il est possible d'utiliser des colonnes de type `DATE` ou `DATETIME` en tant que partitionnement par intervalle et liste (`PARTITION BY RANGE…` et

PARTITION BY LIST…).

La directive PARTITION BY LIST précise les colonnes de partition et liste les valeurs de la clé. Fragmentons les adhérents en deux partitions pour distinguer les hommes des femmes.

Tableau 10-33 Création d'une table partitionnée par liste

Code SQL	Commentaires

`CREATE TABLE Adherent_partition_list`
`(adhid SMALLINT UNSIGNED NOT NULL,`
`nom VARCHAR(25) NOT NULL,`
`prenom VARCHAR(30) NOT NULL,`
`civilite VARCHAR(12) NOT NULL,`
`date_nais DATE NOT NULL, tel VARCHAR(15))`
	Colonnes de la table.

`PARTITION BY **LIST COLUMNS** (civilite)`
`(PARTITION femmes VALUES IN ('Mlle.', 'Mme.'),`
`PARTITION hommes VALUES IN ('Mr.'));`
	Définition de la clé de partition et des partitions.

`INSERT INTO Adherent_partition_list` `SELECT * FROM Adherent;`	Chargement des données dans les partitions.
`CREATE INDEX idx_adh_list_adhid` `ON Adherent_partition_list(adhid);`	Création d'un index local (il n'est pas possible de préciser UNIQUE).

L'optimiseur n'accédera qu'à une seule partition (économie en moyenne de 50 % sur les blocs lus) pour les prédicats filtrant un sexe en particulier. Le plan d'exécution décrit l'utilisation d'une partition et de l'index qui porte sur chaque partition.

```
mysql> EXPLAIN PARTITIONS
     SELECT adhid, nom, tel FROM Adherent_partition_list
          WHERE prenom = 'CELINE'
          AND   (civilite = 'Melle.' OR civilite = 'Mme.')
          AND   adhid IN (19110, 777, 17075);


+----+-------------+-------------------------+------------+-------+--------------------+
| id | select_type | table                   | partitions | type  | possible_keys      |
  key               | key_len | ref  | rows  | Extra                                |
+----+-------------+-------------------------+------------+-------+--------------------+
|  1 | SIMPLE      | Adherent_partition_list | femmes     | range | idx_adh_list_adhid |
  idx_adh_list_adhid | 2       | NULL |    3 | Using where                          |
+----+-------------+-------------------------+------------+-------+--------------------+
```

Cette table est désormais contrainte et il n'est plus question d'ajouter des adhérents dont le sexe est inconnu. L'erreur est explicite : on aurait dû prévoir

une partition pour les extraterrestres.

```
mysql> INSERT INTO Adherent_partition_list
             VALUES (3,'SOUTOU','JEAN','ET.', '1965-05-02','05.31.98.71.02');
ERROR 1526 (HY000): Table has no partition for value from column_list
```

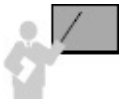
Aussitôt dit, aussitôt programmé : voici l'ajout d'une partition à une table permettant l'insertion de cet adhérent venu de loin.

```
ALTER TABLE Adherent_partition_list
      ADD PARTITION  (PARTITION Extra_terrestres VALUES IN
('ET.'));
```

Il n'existe pas de partition DEFAULT (ce que fait MAXVALUE pour le partitionnement par intervalle) qui se chargerait d'accueillir les lignes ne vérifiant aucune des conditions des partitions existantes.

La clé primaire ou tout index unique doit porter sur la clé de partitionnement (dans notre exemple, la clé n'est en aucun cas unique dans la table). L'erreur retournée le cas échéant est : ERROR **1503** (HY000): A PRIMARY KEY must include all columns in the table's partitioning function**.**

Partitions par hachage



Le partitionnement par hachage permet de répartir les lignes d'une table selon un algorithme interne de hachage en fonction des valeurs de la clé de partitionnement.

La répartition est homogène entre les partitions qui sont de tailles à peu près identiques. Ce mécanisme convient parfaitement lorsqu'aucune sémantique n'intervient dans la clé de partionnement.

La directive PARTITION BY HASH précise les colonnes de partition et dénombre les partitions.

Divisons la table des adhérents en 10 partitions par hachage sur le code

adhérent. En supposant que ce numéro est un entier auto-incrémenté, MySQL gérera les partitions d'une manière cyclique (le premier adhérent ira dans la première partition, le deuxième dans la seconde…, le onzième ira dans la première, etc.).

Tableau 10-34 Création d'une table partitionnée par hachage

Code SQL	Commentaires
`CREATE TABLE Adherent_partition_hash (adhid SMALLINT UNSIGNED NOT NULL, nom VARCHAR(25) NOT NULL, prenom VARCHAR(30) NOT NULL, civilite VARCHAR(12) NOT NULL, date_nais DATE NOT NULL, tel VARCHAR(15))`	Colonnes de la table.
`PARTITION BY HASH (adhid) PARTITIONS 10;`	Définition de la clé de partition et du nombre de partitions.

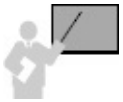
L'objectif de ce type de partitionnement n'est pas un gain de performances pour les requêtes. Il s'avère intéressant pour réduire la taille d'une table en vue d'opérations de maintenance, par exemple.

La clé de partitionnement par hachage ne peut être qu'un entier (ou une expression retournant un entier comme `YEAR (date_nais)`). Attention, toutes les fonctions ne sont pas admises, ainsi la conversion du prénom en hexadécimal, puis en décimal n'est pas permise (`CONV(HEX(prenom),16,10)`).

Si la table dispose d'une clé (primaire ou unique), alors l'expression de partitionnement ne doit faire intervenir que les colonnes de cette clé.

Il existe une variante de cette technique de hachage qui est appelée *linear hash* (la directive de partitionnement devient `PARTITION BY LINEAR HASH`). L'avantage de ce partitionnement est qu'il rend plus performantes les opérations de maintenance (ajout, suppression, fusion, etc.) lorsque les partitions sont de tailles importantes (de l'ordre du téraoctet). En contrepartie, la distribution des données entre les partitions est moins homogène que dans le cas d'un hachage classique.

Partitions par clé



Le partitionnement par clé permet de répartir les lignes d'une table selon un algorithme de hachage (identique à celui de la fonction `PASSWORD()`) basé sur les valeurs de la clé (primaire ou unique) de la table. La répartition est homogène entre les partitions.

La directive `PARTITION BY KEY` précise la liste des colonnes clés (ou aucune et la clé primaire est désignée) de la partition et dénombre les partitions.

S'il n'existe pas de clé primaire mais une clé unique, alors cette dernière peut être utilisée comme clé de partitionnement (à condition qu'une contrainte `NOT NULL` y soit associée).

Le tableau suivant décrit les deux écritures possibles. La table des adhérents est divisée en 10 partitions sur le code adhérent. La table des inscriptions sera répartie selon la même colonne. L'avantage est que la première partition des adhérents sera, en principe, en phase avec la première partition des inscriptions, la deuxième avec la seconde, etc.

Tableau 10-35 Création de tables partitionnées par clé

Partition sur la clé	Partition sur une partie de la clé

`CREATE TABLE Adherent_partition_cle`
`(adhid SMALLINT UNSIGNED PRIMARY KEY,`
`nom VARCHAR(25) NOT NULL,`
`prenom VARCHAR(30) NOT NULL,`
`civilite VARCHAR(12) NOT NULL,`
`date_nais DATE NOT NULL,`
`tel VARCHAR(15))`
`PARTITION BY KEY()`
`PARTITIONS 10;`
	`CREATE TABLE Pratique_partition_cle`
	`(adhid SMALLINT UNSIGNED,`
	`spid TINYINT UNSIGNED,`
	`PRIMARY KEY(adhid,spid))`
	`PARTITION BY KEY(adhid)`
	`PARTITIONS 10;`

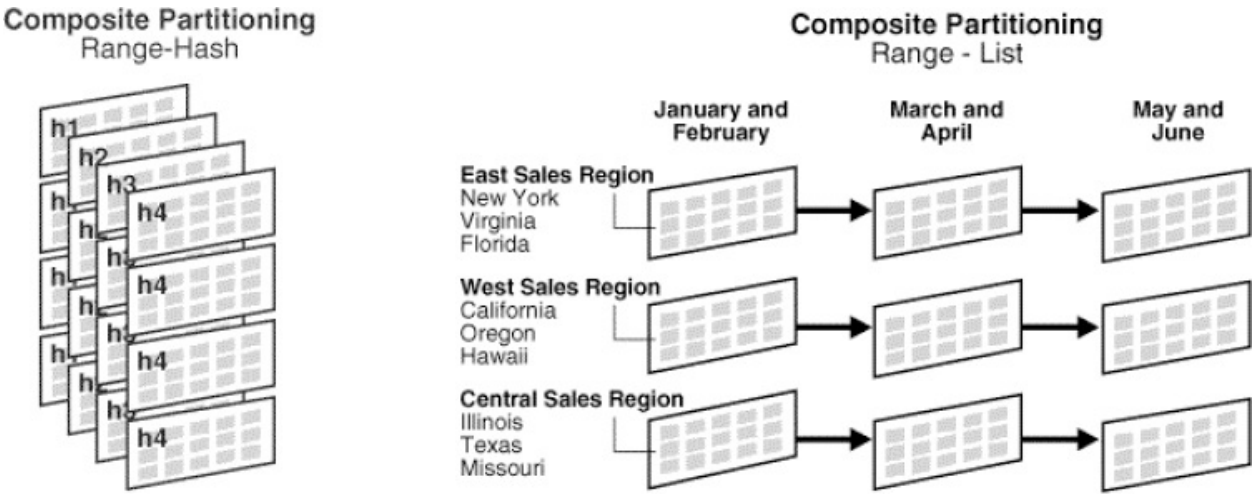
Il existe une variante de cette technique de partitionnement par clé, qui se nomme *linear key* et qui a le même effet que sur la technique de hachage précédemment étudiée. Il s'agit de baser la répartition dans les partitions sur un nombre dérivé de puissances de deux à la place d'un calcul de modulo. Dans

l'exemple précédent, la clause `PARTITION BY LINEAR KEY(adhid)` activerait ce mécanisme.

Sous-partitions

Si ces mécanismes de partitionnement ne sont pas assez précis pour vous, rien ne vous empêche d'utiliser le sous-partitionnement (*composite partinioning*) en divisant encore chaque partition. La figure suivante illustre deux scénarios : à gauche, la table est partitionnée par intervalle de dates et sous-partitionnée par hachage, et à droite, la table est partitionnée par intervalle de dates et sous-partitionnée par liste de régions.

***Figure 10-12** Sous-partitionnements © Document Oracle*



Seules les tables partitionnées par intervalle (`RANGE`) et liste (`LIST`) peuvent être sous-partitionnées. Les deux mécanismes de sous-partitionnement possibles sont le hachage (`HASH`) et la clé (`KEY`).

La stratégie de sous-partitionnement est commune à toutes les partitions (si deux sous-partitions sont définies sur une partition, toutes les partitions principales doivent être divisées en deux).

Supposons que le partitionnement par âge ne suffise pas pour des contraintes de stockage et que l'on désire diviser en deux chaque population. Il est possible

de combiner le partitionnement par intervalle avec un sous-partitionnement par hachage (ici, deux sous-partitions seront définies). La directive SUBPARTITION BY précise le type de sous-partitionnement et SUBPARTITION définit chaque sous-partition.

Tableau 10-36 Création d'une table partitionnée par hachage

Code SQL	Commentaires

`CREATE TABLE Adherent_range_hash`
`(adhid SMALLINT UNSIGNED,`
`nom VARCHAR(25) NOT NULL,`
`prenom VARCHAR(30) NOT NULL,`
	Colonnes de la table.

Code SQL	**Commentaires**
`civilite VARCHAR(12) NOT NULL,`	Définition de la clé de partition.
`date_nais DATE NOT NULL, tel VARCHAR(15))`	Définition de la clé de sous-partitionnement.

`PARTITION BY RANGE COLUMNS` `(date_nais)`
`SUBPARTITION BY HASH``(adhid)`
`( PARTITION retraites`
`VALUES LESS THAN ('1945-01-01')`
`(SUBPARTITION s1, SUBPARTITION s2),`
`PARTITION actifs`
`VALUES LESS THAN ('1993-01-01')`
`(SUBPARTITION s3, SUBPARTITION s4),`
`PARTITION mineurs`
`VALUES LESS THAN (MAXVALUE)`
`(SUBPARTITION s5, SUBPARTITION s6)`
`);`
	Définition des partitions et des deux sous-partitions.

Dictionnaire des données

La vue PARTITIONS de la base INFORMATION_SCHEMA renseigne à propos de toutes les caractéristiques des partitions. La requête suivante permet de retrouver le type de partitionnement et de sous-partitionnement d'une table en particulier (ici, la table précédemment créée).

```
mysql> SELECT DISTINCT PARTITION_METHOD AS "Type part.",
              PARTITION_EXPRESSION AS "Colonne",
              PARTITION_DESCRIPTION "Valeur",
              SUBPARTITION_METHOD AS "Type sous-part.",
              SUBPARTITION_EXPRESSION AS "Colonne"
         FROM INFORMATION_SCHEMA.PARTITIONS
         WHERE TABLE_NAME ='Adherent_range_hash';
+--------------+------------+-------------+-----------------+---------+
| Type part.   | Colonne    | Valeur      | Type sous-part. | Colonne |
+--------------+------------+-------------+-----------------+---------+
| RANGE COLUMNS | `date_nais` | '1945-01-01' | HASH           | adhid   |
| RANGE COLUMNS | `date_nais` | '1993-01-01' | HASH           | adhid   |
| RANGE COLUMNS | `date_nais` | MAXVALUE     | HASH           | adhid   |
+--------------+------------+-------------+-----------------+---------+
```

La requête suivante permet de retrouver pour chaque sous-partition, le nombre de lignes présentes, la taille moyenne des enregistrements et la taille totale.

```
mysql> SELECT PARTITION_NAME AS "PARTITION",
            SUBPARTITION_NAME AS "SOUS-PAR.",
            TABLE_ROWS, AVG_ROW_LENGTH, DATA_LENGTH
            FROM INFORMATION_SCHEMA.PARTITIONS
            WHERE TABLE_NAME ='Adherent_range_hash';
+-----------+----------+------------+----------------+-------------+
| PARTITION | SOUS-PAR. | TABLE_ROWS | AVG_ROW_LENGTH | DATA_LENGTH |
+-----------+----------+------------+----------------+-------------+
| retraites | s1       |       2071 |             94 |      196608 |
| retraites | s2       |       2363 |             83 |      196608 |
| actifs    | s3       |       7770 |            204 |     1589248 |
| actifs    | s4       |       8094 |            196 |     1589248 |
| mineurs   | s5       |       1919 |             85 |      163840 |
| mineurs   | s6       |       2038 |             80 |      163840 |
+-----------+----------+------------+----------------+-------------+
```

Sélections explicites de partitions

Depuis la version 5.6, vous pouvez sélectionner explicitement des partitions (ou sous-partitions). Contrairement au mécanisme de *pruning* qui s'applique aux requêtes en sélectionnant automatiquement les partitions à activer selon les valeurs des enregistrements manipulés, le mécanisme de sélection manuel peut s'appliquer au sein de diverses instructions : `SELECT`, `DELETE`, `INSERT`, `REPLACE`, `UPDATE`, `LOAD DATA` et `LOAD XML`. Le partitionnement explicite se désigne à l'aide de l'option `PARTITION(`*`nom_partition`* `[,…])`.

Considérons la table suivante constituée de trois partitions.

```
CREATE TABLE Adherent_part_explicit
 (adhid     SMALLINT UNSIGNED NOT NULL, nom  VARCHAR(25) NOT NULL,
  prenom    VARCHAR(30) NOT NULL, civilite   VARCHAR(12) NOT NULL,
  date_nais DATE NOT NULL, tel VARCHAR(15), type_adh CHAR(1))
PARTITION BY RANGE COLUMNS (date_nais)
 (PARTITION retraites VALUES LESS THAN ('1945-01-01'),
  PARTITION actifs    VALUES LESS THAN ('1993-01-01'),
  PARTITION mineurs   VALUES LESS THAN (MAXVALUE));
```

Le tableau suivant présente quelques opérations réalisées sur cette table en explicitant les partitions.

Tableau 10-37 Opérations explicites sur les partitions

Instruction SQL	Résultat

`SELECT prenom,nom,date_nais`
`FROM Adherent_part_explicit` **`PARTITION(actifs)`**
`WHERE civilite = 'Mme.'`
`ORDER BY date_nais DESC;`
	Extraction des adhérentes issues de la population active.

`SELECT prenom,nom,tel,date_nais`
`FROM Adherent_part_explicit`
`PARTITION(mineurs, retraites)`
`ORDER BY nom;`
	Extraction de l'identité des adhérents mineurs et retraités.

`UPDATE Adherent_part_explicit` **`PARTITION(mineurs)`** `SET type_adh = 'M';`	Modification pour tous les adhérents mineurs.
`DELETE FROM Adherent_part_explicit` **`PARTITION(retraites);`**	Suppression de tous les adhérents retraités.

Opérations sur les partitions

Les caractéristiques d'une table partitionnée peuvent être modifiées. L'instruction `ALTER TABLE` permet :

- d'ajouter et de supprimer des partitions (`ADD/DROP/TRUNCATE PARTITION`) ;
- de fusionner des partitions (`COALESCE PARTITION`) ;
- de maintenir des partitions après de nombreuses mises à jour (`REORGANIZE/ANALYZE/CHECK/OPTIMIZE/REBUILD/REPAIR PARTITION`) ;
- d'échanger des partitions ou sous-partitions de tables par des tables non partitionnées et inversement (`EXCHANGE PARTITION` depuis la version 5.6).

Restrictions

Il n'est pas possible de partitionner selon une valeur de clé étrangère (*reference-partitionned table*).

Chaque index créé sur une table partitionnée est un index local à la partition (il n'existe pas de mécanisme d'index global). C'est pour cela que la clé primaire (ou tout index unique) d'une table à partitionner doit constituer la clé de partitionnement.

Il n’est pas possible de définir d’index FULLTEXT sur une table partitionnée ni de partitionner une table temporaire.

Le nombre maximal de partitions (sous-partitions incluses) d’une table est limité à 8 192 (en version 5.6).

Annexe

Bibliographie et webographie

Livres

P. BORGHINO, O. DASINI, A. GADAL, *Audit et optimisation MySQL 5*, Eyrolles, 2010.

P. MARTIN, J.PAULI, C. PIERRE DE GEYER, E. DASPET, *PHP 7 avancé*, Eyrolles, 2016.

M. KOFLER, *MySQL 5*, Eyrolles, 2005.

C. SOUTOU, F. BROUARD, *Modélisation de bases de données*, Eyrolles, 2015.

Sites Web

Eyrolles

Éditeur : www.editions-eyrolles.com

Compléments Web : sur la fiche de l'ouvrage sur www.editions-eyrolles.com.

MySQL

Portail : http://www.mysql.com/

Documentations : http://dev.mysql.com/doc/index.html

Manuels de référence de la version *x.y* : http://dev.mysql.com/doc/refman/x.y/en/

Index

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z