eVOGUE

Liam TARDIEU

www.evogue.fr

SUPPORT DE COURS / SQL

Sommaire

➢ SGBD

DEFINITION

Un système de gestion de bases de données (abrégé SGBD) est un ensemble de logiciels qui sert à la manipulation des bases de données. Il sert à effectuer des opérations ordinaires telles que consulter, modifier, construire, organiser, transformer, copier, sauvegarder ou restaurer des bases de données. Il est souvent utilisé par d'autres logiciels ainsi que les administrateurs ou les développeurs.
Autrement dit, c'est un ensemble de logiciels destiné au stockage et à la manipulation de bases de données.

BDD

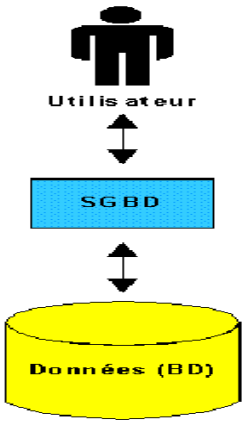
Une base de données « bdd » est un ensemble bien structuré de données relatives à un sujet global. Ces données peuvent être de nature et d'origines différentes.

EXEMPLE

Une banque peut avoir une base de données, qui contient les informations nécessaires sur tous les clients et leurs dépôts d'épargne. Une société d'assurances peut stocker les données relatives aux contrats d'assurances ainsi qu'aux sinistres dans une base de données.
Un système de gestion de bases de données (SGBD) est un programme qui nous permet de créer, de modifier et d'exploiter des bases de données. Ce système constitue donc notre interface pour accéder aux données.

Par analogie:

Un utilisateur utilise un tableur pour accéder aux données d'une feuille de calcul, respectivement un traitement de texte pour accéder au texte d'un document.
Un utilisateur utilise un SGBD pour accéder aux données d'une base de données.



HISTORIQUE

Avant les années 1970, la plupart des systèmes qui permettaient de gérer des grands volumes de données d'une façon plus ou moins cohérente, étaient basés sur de simples fichiers séquentiels. Ces systèmes de gestion de fichiers (SGF) s'avéraient particulièrement limités lorsqu'il s'agissait de gérer une grande masse de données comportant des liens entre elles.

Classiquement, cette masse de données était répartie dans différents fichiers. L'utilisation de ces données n'était possible que par le biais de programmes spécialisés, qui ont dû être réalisés par des programmeurs ayant une connaissance technique approfondie de la structure des fichiers. Chaque nouvelle interrogation du SGF nécessitait donc l'intervention d'un programmeur.

En plus, les SGF n'ont pas assuré la cohérence des données. Le programmeur était seul responsable pour garantir l'intégrité des données. Prenons l'exemple d'un SGF qui était utilisé dans une banque pour la gestion des clients et de leurs dépôts. Rien n'empêchait un programmeur de créer dans le fichier des dépôts un nouveau dépôt pour un client qui n'existait pas du tout dans le fichier des clients, etc.

Vers la fin des années 1960, les premiers systèmes qui étaient capables de cacher la représentation interne des données à l'utilisateur, apparaissaient sur le marché. Dans un système de gestion des dépôts d'une banque, une telle règle pouvait par exemple exprimer le lien explicite entre un dépôt client et une personne. Ces systèmes étaient essentiellement basés sur les deux modèles de données suivants:

- Modèle réseau développé initialement par la "Conference On Data Systems and Languages" (CODASYL) en 1961.
- Modèle hiérarchique développé pour la plus grande partie par la société IBM pendant les années 1965 – 1970.

C'était en 1970 qu'un nouveau modèle, pour représenter les données, le modèle relationnel, fut proposé par E.F.CODD. Le but de ce modèle, était d'accroître l'indépendance vis-à-vis de l'implémentation interne des données. Du point de vue de l'utilisateur, les données sont stockées dans un ensemble de tableaux, appelés "tables relationnelles" ou simplement "tables". Le stockage ainsi que la manipulation des données se basent sur le concept mathématique de l'algèbre relationnelle et du calcul relationnel. Ces concepts proviennent de la théorie mathématique des ensembles, et on y retrouve des notions telles que "Union", "Intersection" ou "Produit cartésien".

Il a fallu attendre le milieu des années 1970 pour voir apparaître les premiers systèmes qui étaient basés sur le modèle relationnel, les "Systèmes de Gestion de Bases de Données Relationnelles (SGBDR)".

En 1976 apparaît le modèle Entité-Association, proposé par P.CHEN, qui donnait aux concepteurs des bases de données relationnelles une méthode adéquate pour modéliser des données d'un domaine quelconque.

Les premiers SGBD qui manipulent des bases de données relationnelles - IBM System R et Oracle V2 - sont apparus en 1978. Pendant les années 1980 et 1990, beaucoup de SGBDR étaient commercialisés pour les différentes plates-formes informatiques.
Aujourd'hui, les bases de données relationnelles jouissent d'une grande popularité. Surtout le domaine de la gestion des données à l'intérieur des entreprises est entièrement dominé par ces systèmes.

Pour la suite de ce cours, nous allons nous limiter à l'étude des bases de données relationnelles. Nous entendons donc par chaque référence à une base de données (bdd), la notion de bases de données relationnelles. Il est également sous-entendu que les deux notions SGBD et SGBDR dénotent un système de gestion de bases de données relationnelles dans le contexte de ce cours.

LES COMPOSANTS

Tables

Les données sont stockées à l'intérieur de tables. Une table peut être comparée à une liste, qui contient des enregistrements relatifs à un sujet bien défini.

Champs

Dans les bases de données, un champ (ou colonne, ou attribut) est la plus petite information d'une table de base de données qui représente en colonne une catégorie d'information, comme un nom ou une adresse. Chaque champ possède un type.

Requêtes

Les requêtes constituent dans un certain sens des "questions" que l'on pose au SGBD. Le résultat d'une requête est toujours un sous-ensemble d'une ou de plusieurs tables.

OPTIMISATION STRUCTURE LOGIQUE

La normalisation permet d'organiser les données afin de limiter les redondances. Les tables sont souvent divisées en plusieurs tables plus petites reliées entre elles par des relations (clés primaires et clés étrangères). L'objectif est d'isoler les données afin que l'ajout, la modification ou l'effacement d'un enregistrement puisse se faire sur une seule table, et se propager au reste de la base par le biais des relations.
Concevoir et normaliser sa base de données correctement, c'est utiliser le modèle 3NF (3rd Normal Form).

OPTIMISATION STRUCTURE PHYSIQUE

Pour optimiser la structure physique de notre base de données, il est essentiel de choisir les bons types de champs.
MySQL supporte un grand nombre de types de champs. Ils peuvent être rassemblés en trois catégories : les types numériques, temporels et chaînes de caractères. (*Un chapitre y est consacré à la fin du support)*

➢ Les tables

DEFINITION

Une table est une collection de données relatives à un sujet bien défini, par exemple les employés d'une société ou les livres d'une bibliothèque.

Elle contient des enregistrements dont chacun est composé par les mêmes champs de données.

Voici, à titre d'exemple, quelques employés d'une société:

Nom : Grand
Prénom : Fabrice
Age : 35
Salaire : 24000K€
Service : comptabilité

Nom : Durand
Prénom : Damien
Age : 27
Salaire : 16000K€
Service : commercial

Nom : Vignal
Prénom : Mathieu
Age : 43
Salaire : 28000K€
Service : informatique

Voici la table « employes » nécessaire pour stocker les informations concernant ces employés dans une base de données.

Nom	Prenom	Age	Salaire	Service
Grand	Fabrice	35	24000	Comptabilité
Durand	Damien	27	16000	Commercial
Vignal	Mathieu	43	28000	Informatique

Propriétés des tables:

- Les champs de données définissent les informations, que l'on souhaite stocker dans la table (ex: des informations concernant les employés d'une société).
- Chaque enregistrement représente une occurrence de ce que l'on stocke (ex: un employé).
- Chaque table possède un nom unique (ex: employes).
- Chaque enregistrement correspond à une ligne de la table.
- Chaque champ correspond à une colonne de la table.
- Chaque champ peut représenter des données de nature différente (Nom, Salaire, Date de naissance …).
- Chaque champ peut représenter des données de type différent (Texte, Nombres, Dates …).

➢ Clé primaire

CAS D'ETUDE

Exemple avec une base de données nommée « taxis ».

Une table reprenant les données concernant les voitures d'une société de taxis contient par exemple pour chaque enregistrement les informations suivantes:

- Marque
- Modèle
- Cylindrée
- Poids

Il est évident que les informations sont de types différents. Tandis que la marque, le modèle et la couleur sont représentés par des chaînes de caractères (ex : "Ford", "BMW", …), la cylindrée est représentée par des valeurs numériques.

Marque	Modèle	Cylindrée	Couleur
BMW	525i	2500	noir
FORD	Orion	1800	gris
BMW	320i	2000	blanc

PROBLEMATIQUE

Dans la plupart des cas, nous désirons pouvoir identifier de manière unique chaque enregistrement de la table. Ceci n'est pas possible pour notre table avec les taxis. Il se peut très bien que le propriétaire de la société achète par exemple une deuxième BMW 320i, qui possède aussi une cylindrée de 2000 ccm et que le véhicule soit de couleur blanche. Dans ce cas, nous avons 2 enregistrements complètement identiques dans notre base de données. Cela nous empêche d'identifier clairement un des 2 enregistrements.

Il nous faut donc un moyen, qui nous permette d'adresser sans ambiguïté chaque enregistrement dans la table →**une clé primaire** !

La clé primaire nous permet d'identifier de manière unique chaque enregistrement d'une table.

Examinons notre cas de la société de taxis. Aucun des 4 champs seuls, et aucune combinaison des 4 champs ne se prêtent comme candidats pour devenir clé primaire, car aucun de ces champs ne contient des valeurs uniques à un et un seul taxi. Supposons par exemple la marque et le modèle comme clé primaire. Au cas où la société achèterait une deuxième BMW 320i, nous ne pourrions plus distinguer les deux voitures.

EXPLICATIONS

Le ou les champs, qui forme(nt) la clé primaire doivent impérativement avoir des valeurs qui sont uniques pour toute la table, et qui permettent donc d'identifier chaque enregistrement.

Exemple:

Le numéro de matricule pour les assurés des Caisses de maladie.

Le numéro client pour les clients d'une vidéothèque.

SOLUTIONS

En ce qui concerne les taxis, nous avons deux possibilités:

1. Analyser s'il n'existe pas d'informations concernant les taxis qui ne soient pas encore stockées dans la table et qui feraient une clé primaire valable. Une telle information serait par exemple le numéro de châssis ou la plaque d'immatriculation unique pour chaque voiture. Nous pourrions donc ajouter un champ *num_chassis* et définir ce champ comme clé primaire. Ceci a comme désavantage que le numéro de châssis d'une voiture est un numéro assez long et compliqué, ce qui défavorise une utilisation conviviale de la table.
2. Nous pourrions inventer un numéro de taxi allant simplement de 1 jusqu'au nombre de taxis que la société possède. Le premier taxi enregistré serait le numéro TAXI=1, le deuxième le numéro TAXI=2, etc. Bien que ce numéro n'ait aucune signification réelle, cette méthode de création de clés primaires artificielles est très répandue, et la plupart des SGBD offre même un type de données prédéfini pour générer des valeurs uniques pour de telles clés primaires. Notre table aurait dans ce cas-là la structure suivante:

Id_taxis	Marque	Modèle	Cylindrée	Couleur
1	BMW	525i	2500	noir
2	FORD	Orion	1800	gris
3	BMW	320i	2000	blanc

Le nom du champ qui sert de clé primaire est : **id_taxis**

Cette convention de nom est fréquemment utilisée.

➢ Clé étrangère

DEFINITION

Un champ qui, dans une table, fait référence à la clé primaire d'une autre table est appelé clé étrangère (anglais: foreign key). Ainsi sont définies les relations entre les tables.

CAS D'ETUDE

Nous l'avons dit, une base de données bien constituée est rarement composée d'une seule table, mais d'un ensemble de tables, entre lesquelles il existe certaines relations. Nous reprenons l'exemple précédent (*sur les clés primaires*) et observons la table nommée « voiture » sur la base de données nommée « taxis » :

Id_taxis	Marque	Modèle	Cylindrée	Couleur
1	BMW	525i	2500	noir
2	FORD	Orion	1800	gris
3	BMW	320i	2000	blanc

La table « voiture » ci-dessus nous donne des informations sur les véhicules de la société (d'autres renseignements sont possibles tel que : le nombre de Kms, le nombre de chevaux, , etc.).
Jusqu'à présent, nous n'avions aucune information sur les conducteurs de ces taxis ;
Observons maintenant la table « conducteur » sur la base de données nommée « taxis » :

Id_conducteur	Nom	Prénom
1	Desprez	Thierry
2	Sennard	Emilie
3	Collin	Jean-Louis

La table « conducteur » contient certaines informations concernant les conducteurs des véhicules mais pas le nom du véhicule qu'ils conduisent actuellement. Les informations sur les véhicules se trouvent dans la table voiture. Cependant dans la table conducteur se trouve le champ id_conducteur et dans la table voiture se trouve le champ id_taxis (qui donne un indice vers le conducteur du véhicule).Nous pouvons donc facilement relier les deux informations afin de retrouver le nom du conducteur pour chaque véhicule (ainsi nous savons que Thierry Desprez conduit la BMW 525i).Conventionnellement, on dit que id_conducteur est une clé étrangère, qui fait référence à la clé primaire id_taxis de la table voiture.

MODELISATION

Coté modélisation et pour aller jusqu'au bout de cet exemple, il nous faudrait naturellement faire une table de jointure afin de faire des associations entre les véhicules (taxis) et les personnes physiques (conducteurs). Dès lors que l'on s'intéresse aux cardinalités, nous devons imaginer un système ou chaque conducteur puisse conduire plusieurs véhicules (taxis).

Id_conducteur	Id_taxis
1	3
2	2
3	1

➢ Index

TYPE : UNIQUE, SPATIAL, FULLTEXTE

Plusieurs types d'index existent.

- Index unique : permet d'éviter 2 mêmes valeurs au sein du même champ (colonne).
  Exemple: Sur un site, deux utilisateurs ne peuvent pas avoir le même pseudo, il serait donc judicieux d'utiliser l'index unique sur le champ (colonne) « pseudo »
- Index : Permet de mettre en place les contraintes d'intégrités.
- Index Spatial : Permet d'optimiser les calculs impliquant des positionnements ou des distances. exemple: Des coordonnées GPS.
- Index FullTexte : Permet d'optimiser les recherches sur cette colonne.

EXEMPLE

Une des utilisations fréquentes des tables consiste dans la recherche et le tri des enregistrements. Lorsque les tables contiennent un grand nombre d'enregistrements, la recherche de certains enregistrements ainsi que le tri d'enregistrements nécessitent de plus en plus de temps. Les index sont des structures qui accélèrent les tris et recherches dans les tables, ainsi que l'exécution de certaines requêtes.

Reprenons notre exemple des employés d'une société. Une recherche intéressante serait par exemple: *MONTRE-MOI TOUS LES EMPLOYES DU SERVICE INFORMATIQUE !*

Il serait aussi intéressant de trier les employés par leur nom de famille. Au cas où la table contiendrait beaucoup d'enregistrements, nous devrions d'abord créer un index sur le champ « Nom », afin d'accélérer le tri.

Créer par exemple un index sur le champ « Nom » veut dire que le SGBD copie toutes les valeurs existantes du champ « Nom » dans une liste spéciale à 2 colonnes. La deuxième colonne contient les noms triés par ordre alphabétique, et la première contient une référence vers l'enregistrement correspondant de la table.

Il est évident que par la suite de la création de cet index, toutes les recherches et les tris concernant le nom de l'employé sont accélérés, puisque le SGBD consulte uniquement l'index pour retrouver le bon nom, pour ensuite utiliser la référence de l'index vers l'enregistrement correspondant de la table.

PROPRIETES

- Un index est toujours lié à un ou plusieurs champs d'une table.
- Un index peut seulement contenir des champs ayant un des types de données Texte, Numérique ou Date/Heure.
- Un index est automatiquement mis à jour par le SGBD lors d'un ajout, d'une modification ou d'une suppression d'enregistrements dans la table. Ceci est transparent pour l'utilisateur de la BDD.

Les champs référencés fréquemment dans les recherches et tris doivent être indexés. Définir trop d'index sur une table ralentit en général les opérations d'ajout, de modification et de suppression, parce que le SGBD doit mettre à jour la table et l'index.

La clé primaire est toujours indexée à l'aide d'un index sans doublons (unicité).

➢ Le langage SQL

GENERALITES

Il faut d'abord formuler une requête et puis l'exécuter afin d'avoir des résultats. Vous pouvez probablement bien vous imaginer que les SGBD actuels ne comprennent pas le langage naturel. Aucun SGBD n'offre une possibilité d'écrire par exemple « ***Je veux voir tous les employés du service commercial*** » Pour formuler une requête, l'utilisateur doit donc utiliser un langage spécialisé pour ce domaine. Nous distinguons deux catégories au sein du langage SQL :

Le langage de définition des données, qui permet de décrire les tables et autres objets manipulés par le SGBDR; et le langage de manipulation des données, qui permet de modifier les informations contenues dans la base.

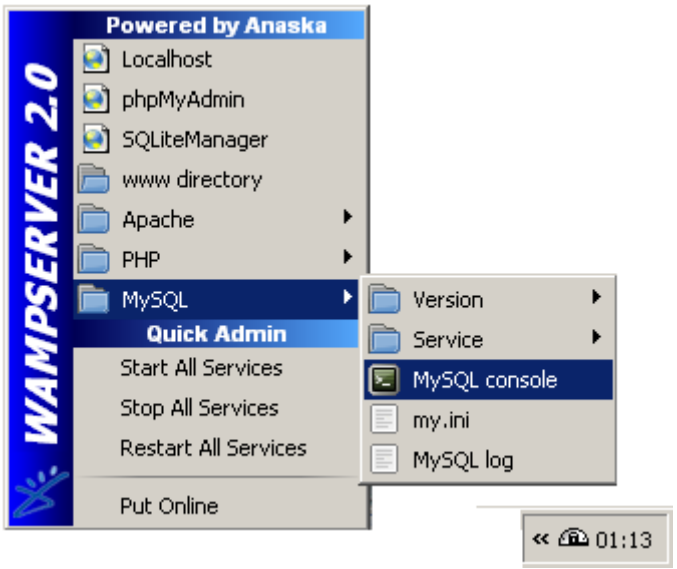
DEFINITION

Le langage **SQL (Structured Query Language)** est un **standard international**, en ce qui concerne les langages de manipulation des bases de données. SQL est connu par tous les SGBDR. Il faut cependant mentionner que, malgré la présence de standards internationaux tels que SQL-86, SQL-89 ou SQL-92, chaque SGBD sur le marché utilise un peu son propre dialecte du langage SQL.

INTRODUCTION

Nous allons nous servir de la console mysql.

Pour cela il faut avoir lancé wampserver et cliqué sur l'icône dans la barre des tâches :



➢ Définitions des données

LES COMPTES/DROITS

Pour MySQL, les utilisateurs n'existent qu'en fonction de l'ordinateur d'où ils tentent de se connecter... autrement dit, si vous avez créé user@localhost, vous ne pouvez pas l'utiliser pour vous connecter à distance. Si l'utilisateur root@localhost ne nous convient pas, nous créons un nouvel utilisateur « test » qui a tous les droits sur la base tic_entreprise.

Créer un utilisateur

```
CREATE USER test;
```

Voir les utilisateurs

```
mysql> SELECT user, host FROM mysql.user;
+------+-----------+
| user | host      |
+------+-----------+
| root | localhost |
+------+-----------+
```

Attributions des droits

```
mysql> GRANT ALL ON nomdelabdd.*  TO test@'%' IDENTIFIED BY 'test';
```

Révocation des droits

```
REVOKE ALL ON nomdelabdd.* FROM test;
```

Voir les utilisateurs

```
mysql> SELECT user, host FROM mysql.user;
+------+-----------+
| user | host      |
+------+-----------+
| test | %         |
| root | localhost |
+------+-----------+
```

Supprimer un utilisateur

```
mysql> DROP user test;
```

LES BASES DE DONNEES

En informatique, une base de données (Abrégé. : « BD » ou « BDD ») est un lot d'informations stockées dans un dispositif informatique. Les technologies existantes permettent d'organiser et de structurer la base de données de manière à pouvoir facilement manipuler le contenu et stocker efficacement de très grandes quantités d'informations.

Voir les bases de données

Suivant les versions de MySQL différentes bases de données sont déjà installées. Pour en connaître le nom :

```
mysql> SHOW DATABASES ;
+--------------------+
| Database           |
+--------------------+
| information_schema |
| mysql              |
+--------------------+
```

Par exemple la base de données «information_schema» est présente pour recenser des informations telles que les utilisateurs et autres…

Créer une nouvelle base de données

```
mysql> CREATE DATABASE tic_entreprise;
```

Observer à nouveau les bases de données

```
mysql> SHOW DATABASES ;
+--------------------+
| Database           |
+--------------------+
| information_schema |
| mysql              |
| tic_entreprise     |
+--------------------+
```

Utiliser une base de données

```
mysql> USE tic_entreprise;
Database changed
```

Cela signifie que nous allons travailler (envoyer des requêtes) sur la base entreprise.

Supprimer une base de données

```
mysql> DROP DATABASE tic_entreprise ;
```

LES TABLES

Création d'une table

```
CREATE  TABLE  Nom_de_la_table
          ( nom_champ1  type_ de_données  [Not null / Null] ,
           nom_champ1  type_ de_données  [Not null / Null] ,
           …
          ) ;
```

Exemple (sur la bdd tic_entreprise) :

```
mysql>
CREATE TABLE `employes` (
  `id_employes` int(4) NOT NULL,
  `prenom` varchar(20) default NULL,
  `nom` varchar(20) default NULL,
`sexe` enum('m','f') NOT NULL,
  `service` varchar(30) default NULL,
  `date_embauche` date default NULL,
  `salaire` float default NULL,
  `id_secteur` int(2) NOT NULL,
  PRIMARY KEY  (`id_employes`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1;
```

Modification d'une table

```
ALTER TABLE  nom_de_la_table
```

Ajout d'une colonne:

```
ALTER TABLE  nom_de_la_table ADD
( nom_de_la_colonne1 type_de_données  [Not Null / Null] ,
  nom_de_la_colonne2 type_de_données  [Not Null / Null] …
) ;
```

Modification d'une colonne existante

```
ALTER TABLE  nom_de_la_table MODIFY
( nom_de_la_colonne1 type_de_données  [Not Null / Null] ,
  nom_de_la_colonne2 type_de_données  [Not Null / Null] ,
 …
) ;
```

Supprimer une table

```
mysql> DROP TABLE nom_de_la_table ;
```

Vider une table

```
mysql> TRUNCATE nom_de_la_table;
```

Observer la structure de la table ainsi que les champs

```
mysql> DESC  employes;
```

➢ Manipulation des données

DEFINITION

Nous avons vu que la plupart des SGBD offre la possibilité d'effectuer des recherches directement dans les tables. Les possibilités de formuler des critères de recherche sont cependant souvent assez limitées. Heureusement, la plupart des SGBD nous offre également la possibilité de poser pratiquement n'importe quelle "question" à nos tables, sous forme de requête.

Les requêtes servent donc à répondre aux questions basées sur le contenu d'une ou de plusieurs tables. Nous allons plus tard étudier des requêtes, qui se basent sur plusieurs tables, mais pour l'instant nous allons nous limiter aux questions simples basées sur une seule table.

Il existe 4 types de requêtes :

- requête de sélection,
- requête d'insertion,
- requête de modification,
- requête de suppression.

Pour chaque requête nous retrouvons le cycle suivant :

REQUETES DE SELECTION

Une requête de sélection serait par exemple:

```
SELECT <Nom des champs>
FROM <Nom de la table>
WHERE <Critères de sélection>;
```

Par convention, dans une requête, les mots-clés s'inscrivent en majuscule afin de les distinguer des mots variables. Ceci étant, les requêtes ne sont pas sensibles à la casse.

Langage Naturel (question) :

- Quels sont les noms et prénoms des employés travaillant dans l'entreprise ?

Langage SQL (requête) :

```
mysql> SELECT nom, prenom FROM employes ;
```

Résultats :

```
+----------+-------------+
| nom      | prenom      |
+----------+-------------+
| chevel   | daniel      |
| cottet   | julien      |
| grand    | fabrice     |
| fellier  | elodie      |
| lafaye   | stephanie   |
| durand   | damien      |
| winter   | thomas      |
| blanchet | laura       |
| laborde  | jean-pierre |
| desprez  | thierry     |
| sennard  | emilie      |
| perrin   | celine      |
| collier  | melanie     |
| perrin   | chloe       |
| miller   | guillaume   |
| martin   | nathalie    |
| lagarde  | benoit      |
| vignal   | mathieu     |
| thoyer   | amandine    |
+----------+-------------+
```

Le résultat est un sous-ensemble de la table avec seulement les noms et prénoms des employés recensés dans l'entreprise. Pour chacun de ces enregistrements, le SGBD affiche en plus seulement les champs explicitement demandés appartenant à ce sous-ensemble (Nom et Prénom).

Nous appelons ces requêtes "Requêtes de Sélection", puisqu'il s'agit d'une sélection de certains enregistrements.

Langage Naturel (question) :

- Quels sont les différents services occupés par les employés travaillant dans l'entreprise ?

Langage SQL (requête) :

```
mysql> SELECT service FROM employes;
```

Résultats

```
+-------------------------+
| service                 |
+-------------------------+
| informatique            |
| secretariat             |
| comptabilite            |
| secretariat             |
| assistant manager       |
| commercial              |
| commercial              |
| direction               |
| direction               |
| standardiste            |
| commercial              |
| commercial              |
| commercial              |
| commercial              |
| commercial              |
| juridique               |
| chef de projet          |
| informatique            |
| charge de communication |
+-------------------------+
```

Cette requête renvoie les services qu'occupent les employés conformément à ce que nous lui avons demandé, cependant il n'y a pas qu'un employé par service, il y a donc des doublons dans la liste de résultats.

Langage SQL (requête) :

```
mysql> SELECT DISTINCT service FROM employes;
```

Résultats

```
+-------------------------+
| service                 |
+-------------------------+
| informatique            |
| secretariat             |
| comptabilite            |
| assistant manager       |
| commercial              |
| direction               |
| standardiste            |
| juridique               |
| chef de projet          |
| charge de communication |
+-------------------------+
```

Requête avec un distinct qui évite les doublons ! Cette requête sélectionne les différents services occupés par les employés

Langage Naturel (question) :

- Je souhaite connaître toutes les informations des employés travaillant dans l'entreprise ?

Langage SQL (requête) :

```
mysql> SELECT id_employes, prenom, nom, sexe, service, date, salaire, id_secteur FROM employes;
```

Ou alors :

```
mysql> SELECT * FROM employes ;
```

L'opérateur ***** permet d'afficher tous les champs définis dans la table, cela évite d'avoir à les répéter.

CRITERES DE SELECTION

Les critères de sélection sur une requête de sélection permettent d'ajouter une condition dans la formulation de la requête. Le plus souvent avec le mot « **where** ».

Langage Naturel (question) :

- Je souhaite connaître le nom et prénom de tous les employés de l'entreprise travaillant dans le service informatique ?

Langage SQL (requête) :

```
mysql> SELECT nom, prenom FROM employes WHERE service='informatique';
```

Résultats

```
+--------+---------+
| nom    | prenom  |
+--------+---------+
| chevel | daniel  |
| vignal | mathieu |
+--------+---------+
```

Langage Naturel (question) :

- Je souhaite connaître le nom, le prénom ainsi que la date d'embauche de tous les employés de l'entreprise recrutés entre 2006 et 2010 ?

Langage SQL (requête) :

```
mysql> SELECT nom, prenom, date_embauche FROM employes WHERE date_embauche BETWEEN
        '2006-01-01' AND '2010-12-31';
```

Resultats

```
+---------+----------+---------------+
| nom     | prenom   | date_embauche |
+---------+----------+---------------+
| chevel  | daniel   | 2010-07-05    |
| cottet  | julien   | 2007-01-18    |
| winter  | thomas   | 2006-05-01    |
| desprez | thierry  | 2009-11-17    |
| perrin  | celine   | 2006-09-10    |
| thoyer  | amandine | 2010-01-23    |
+---------+----------+---------------+
```

Langage Naturel (question) :

- Je souhaite connaître le prénom des personnes commençant par la lettre « s » dans l'entreprise?

Langage SQL (requête) :

```
mysql> SELECT prenom FROM employes WHERE prenom LIKE 's%';
```

Résultats

```
+-----------+
| prenom    |
+-----------+
| stephanie|
+-----------+
```

Langage Naturel (question) :

- Je souhaite connaître le prénom des personnes de l'entreprise qui contient un trait d'union dans leur prénom ?

Langage SQL (requête) :

```
mysql> SELECT prenom FROM employes WHERE prenom LIKE '%-%';
```

Langage Naturel (question) :

- Je souhaite afficher le prénom des personnes de l'entreprise commençant par la lettre « d » et qui est composé de 5 lettres tout en finissant par un « l ».

Langage SQL (requête) :

```
mysql> SELECT prenom FROM employes WHERE prenom LIKE 'd____l';
```

Explications

Le % représente une séquence de zéros ou de plusieurs caractères.

Le _ représente une séquence de caractères quelconques.

Exemple:

Le filtre 'BL__' sélectionne par exemple les valeurs 'BLEU' ou 'BLUE' mais pas 'BLANC' (car il y a une lettre supplémentaire).

OPERATEURS DE COMPARAISONS

Les critères de sélection constituent une expression logique qui peut prendre la valeur 'Vrai' ou 'Faux'. Voici les opérateurs de comparaison:

- **=** "est égal"
- **>** "strictement supérieur"
- **<** "strictement inférieur"
- **>=** "supérieur ou égal"
- **<=** "inférieur ou égal"
- **<> ou !=** "est différent"

Langage Naturel (question) :

- Je souhaite connaître le nom et prénom de tous les employés de l'entreprise NE travaillant PAS dans le service informatique ?

Langage SQL (requête) :

```
mysql> SELECT nom, prenom FROM employes WHERE service <> 'informatique';
```

Ou alors :

```
mysql> SELECT nom, prenom FROM employes WHERE service != 'informatique';
```

Langage Naturel (question) :

- Je souhaite connaître le nom et prénom, des employés de l'entreprise ayant un salaire supérieur à 3000€ ?

Langage SQL (requête) :

```
mysql> SELECT nom, prenom, service, salaire FROM employes WHERE salaire>3000;
```

Résultats

```
+----------+-------------+-----------+---------+
| nom      | prenom      | service   | salaire |
+----------+-------------+-----------+---------+
| blanchet | laura       | direction |    3050 |
| laborde  | jean-pierre | direction |    5000 |
| martin   | nathalie    | juridique |    3200 |
+----------+-------------+-----------+---------+
```

Langage Naturel (question) :

- Je souhaite connaître le nom et prénom de l'employé de l'entreprise ayant le salaire le plus élevé (nous ne le connaissons pas).

Langage SQL (requête) :

```
mysql> SELECT nom, prenom, service, salaire FROM employes ORDER BY salaire DESC LIMIT 0,1;
```

Résultats

```
+---------+-------------+-----------+---------+
| nom     | prenom      | service   | salaire |
+---------+-------------+-----------+---------+
| laborde | jean-pierre | direction |    5000 |
+---------+-------------+-----------+---------+
```

OPERATEURS ARITHMETIQUES

Les opérateurs arithmétiques sont pour les plus connus le « **+** » pour l'addition, le « **-** » pour la soustraction, le « ***** » pour la multiplication et le « **/** » pour la division.

Langage Naturel (question) :

- Je souhaite connaître le nom, le prénom et le salaire **annuel** de tous les employés dans l'entreprise ?

Langage SQL (requête) :

```
mysql> SELECT nom, prenom, salaire*12 AS salaire_annuel FROM employes;
```

Dans la table « employes » de notre base de données tic_entreprise, le montant des salaires est exprimé comme prix mensuel. Afin de ramener leur salaire mensuel sur leur salaire annuel, il faut faire une opération « ***12** » (car il y a 12 mois dans une année). Le mot « **as** » nous permet de renommer la colonne (***cela donne un alias uniquement pour la visibilité du résultat***) de manière à ne pas faire de confusion avec le salaire mensuel.
Nous pouvons également trier ces résultats par ordre croissant ou décroissant en ajoutant « order by salaire asc », nous y reviendrons plus loin dans le support :

Résultats

nom	prenom	salaire_annuel
chevel	daniel	20400
cottet	julien	14040
grand	fabrice	19200
fellier	elodie	15000
lafaye	stephanie	21300
durand	damien	15000
winter	thomas	30600
blanchet	laura	36600
laborde	jean-pierre	60000
desprez	thierry	13200
sennard	emilie	21600
perrin	celine	18000
collier	melanie	22800
dubar	chloe	25200
miller	guillaume	20400
martin	nathalie	38400
lagarde	benoit	24600
vignal	mathieu	21600
thoyer	amandine	18000

Résultats (avec tri par ordre croissant)

nom	prenom	salaire_annuel
desprez	thierry	13200
cottet	julien	14040
fellier	elodie	15000
durand	damien	15000
perrin	celine	18000
thoyer	amandine	18000
grand	fabrice	19200
chevel	daniel	20400
miller	guillaume	20400
lafaye	stephanie	21300
vignal	mathieu	21600
sennard	emilie	21600
collier	melanie	22800
lagarde	benoit	24600
dubar	chloe	25200
winter	thomas	30600
blanchet	laura	36600
martin	nathalie	38400
laborde	jean-pierre	60000

Pour compter des enregistrements selon certains critères nous utiliserons le mot : **count**

Pour calculer la somme des valeurs d'un champ indiqué (pourvu que cette valeur soit différente de NULL), nous utiliserons le mot : **SUM**

Pour calculer la moyenne des valeurs des champs indiqués (pourvu que cette valeur soit différente de NULL), nous utiliserons le mot : **AVG**

Pour déterminer la plus grande des valeurs indiquées (pourvu que cette valeur soit différente de NULL), nous utiliserons le mot : **MAX**

A l'inverse, le mot sera : **MIN**

Langage Naturel (question) :

- Je souhaite connaître la masse salariale de l'entreprise (sur une année) ?

Langage SQL (requête) :

```
mysql> SELECT SUM(salaire*12) FROM employes;
```

Résultats

```
+-----------------+
| sum(salaire*12) |
+-----------------+
|          455940 |
+-----------------+
```

Langage Naturel (question) :

- Je souhaite savoir quel est le salaire le plus bas que j'attribue dans l'entreprise (sur un mois) ?

Langage SQL (requête) :

```
mysql> SELECT MIN(salaire) FROM employes;
```

Résultats

```
+---------------+
| min(salaire)  |
+---------------+
|         1100  |
+---------------+
```

Langage Naturel (question) :

- Je souhaite savoir quel est le salaire le plus bas que j'attribue dans l'entreprise ainsi que le prénom de l'employé concerné ?

Langage SQL (requête) :

```
mysql> SELECT prenom,salaire FROM employes WHERE salaire = (SELECT MIN(salaire) FROM
       employes);
```

Résultats

```
+---------+---------+
| prenom  | salaire |
+---------+---------+
| thierry |    1100 |
+---------+---------+
```

Langage Naturel (question) :

- Je souhaite connaitre le salaire moyen dans l'entreprise ?

Langage SQL (requête) :

```
mysql> SELECT ROUND(AVG(salaire)) FROM employes;
```

Résultats

```
+---------------------+
| round(avg(salaire)) |
+---------------------+
|                2000 |
+---------------------+
```

La fonction prédéfinie round() permet d'arrondir le résultat.

Langage Naturel (question) :

- Je souhaite connaître le nombre de femmes travaillant au sein de l'entreprise ?

Langage SQL (requête) :

```
mysql> SELECT COUNT(sexe) FROM employes WHERE sexe='f';
```

Résultats

```
+-------------+
| count(sexe) |
+-------------+
|       9     |
+-------------+
```

OPERATEURS LOGIQUES

Il existe plusieurs opérateurs logiques:

NOT (Négation logique)

L'opérateur NOT inverse le résultat d'une expression logique.

AND (Et, logique)

L'opérateur AND nous permet de combiner plusieurs conditions dans une expression logique.
L'expression logique retourne uniquement la valeur 'Vrai' lorsque toutes les conditions sont remplies.

OR (Ou logique)

L'opérateur OR nous permet de combiner plusieurs conditions dans une expression logique.
L'expression logique retourne la valeur 'Vrai' lorsqu'au moins une des conditions est remplie.

IN (Dans)

L'opérateur IN (<Liste de valeurs>) permet de déterminer si la valeur d'un champ donné appartient à une liste de valeurs prédéfinies.
Les valeurs dans la liste des valeurs sont généralement des valeurs numériques, des valeurs du type Texte ou des valeurs du type Date.

Généralement, chaque champ défini dans une table possède une valeur bien définie. Il existe pourtant des situations dans lesquelles cette valeur est inconnue ou n'existe pas temporairement.

Les SGBD nous offrent en général 3 valeurs pour ces types de situation :

- Le nombre 0 ;
- La chaîne de caractères vide ('') ;
- La valeur prédéfinie NULL (Valeur indéterminée, Champ vide).

Voici, ci-dessous, des exemples pour chacun des opérateurs énoncés ici :

Langage Naturel (question) :

- Je souhaite connaître le prénom des employés travaillant dans le service comptabilité et le service informatique ?

Langage SQL (requête) :

```
mysql> SELECT prenom,service FROM employes WHERE service IN ('comptabilite', 'informatique');
```

Résultats

```
+---------+--------------+
| prenom  | service      |
+---------+--------------+
| daniel  | informatique |
| fabrice | comptabilite |
| mathieu | informatique |
+---------+--------------+
```

A l'inverse, pour connaître le prénom des employés ne faisant pas partie des services comptabilité et informatique, nous pouvons exprimer la requête suivante :

```
mysql> SELECT prenom,service FROM employes WHERE service NOT IN ('comptabilite',
        'informatique');
```

Pour que cela soit plus clair, nous pouvons trier ce résultat par ordre croissant (asc) ou décroissant (desc) :

```
mysql> SELECT prenom,service FROM employes WHERE service NOT IN ('comptabilite',
        'informatique') ORDER BY service ASC;
```

Résultats

```
+-------------+--------------------------+
| prenom      | service                  |
+-------------+--------------------------+
| julien      | secretariat              |
| elodie      | secretariat              |
| stephanie   | assistant manager        |
| damien      | commercial               |
| thomas      | commercial               |
| laura       | direction                |
| jean-pierre | direction                |
| thierry     | standardiste             |
| emilie      | commercial               |
| celine      | commercial               |
| melanie     | commercial               |
| chloe       | commercial               |
| guillaume   | commercial               |
| nathalie    | juridique                |
| benoit      | chef de projet           |
| amandine    | charge de communication  |
+-------------+--------------------------+
```

Langage Naturel (question) :

- Je souhaite connaître le prénom et nom des employés travaillant dans le service commercial avec un salaire inférieur ou avoisinant les 1500€ ?

Langage SQL (requête) :

```
mysql> SELECT prenom, nom, salaire, service FROM employes WHERE service='commercial' AND
        salaire<= 1500 ;
```

Résultats

```
+--------+--------+---------+------------+
| prenom | nom    | salaire | service    |
+--------+--------+---------+------------+
| damien | durand |    1250 | commercial |
| celine | perrin |    1500 | commercial |
+--------+--------+---------+------------+
```

Langage Naturel (question) :

- Je souhaite connaître le prénom et nom des employés du service commercial travaillant dans le secteur 10 ou le secteur 20 ?

Langage SQL (requête) :

```
mysql> SELECT prenom, nom, service, id_secteur FROM employes WHERE service='commercial' AND
        (id_secteur= 10 OR id_secteur=40) ;
```

Résultats

```
+----------+--------+------------+-----------+
| prenom   | nom    | service    | id_secteur |
+----------+--------+------------+-----------+
| thomas   | winter | commercial |      20   |
| celine   | perrin | commercial |      10  |
| guillaume | miller | commercial |      20  |
+----------+--------+------------+-----------+
```

La présence des parenthèses dans la requête a une incidence sur le résultat.

REQUETES D'INSERTION

INSERT INTO <Nom de la table>(<Liste des champs>) VALUES (<Valeurs pour les champs>);

Les requêtes d'insertion servent à insérer des données dans la table. Par exemple, si nous venons de recruter un nouvel employé à la date du 28/01/2012, nous allons devoir l'insérer dans notre table « **employés** » pour le répertorier avec les autres. (*toujours dans la bdd **tic_entreprise***) :

```
mysql> INSERT INTO `employes` (`id_employes`, `prenom`, `nom`, `sexe`, `service`, `date_embauche`, `salaire`, `id_secteur`) VALUES (7259, 'alexis', 'richy', 'm', 'informatique', '2011-12-28', 1800, 10);
```

Si la console mysql n'a pas retourné d'erreur, nous pouvons contrôler la présence du nouvel employé dans la table par une simple requête de sélection/affichage (*vu dans un chapitre précédent*).
Il est également possible d'effectuer une requête d'insertion sans préciser les noms des champs avant les valeurs. Il faut cependant faire attention à l'ordre des valeurs qui seront en corrélation avec l'ordre des champs dans la table.

Exemple

```
mysql> INSERT INTO employes VALUES  (7259, 'alexis', 'richy', 'm', 'informatique', '2012-01-28', 1800,
       10)
```

Remarques

- Les valeurs sont séparées par des virgules.
- En SQL, les données sont entourées d'apostrophes.
- Les dates sont indiquées dans le format américain Année/Mois/Jour
- C'est le contraire du format français : Jour/Mois/Année

Créons maintenant l'exemple avec la table « **localite** » sur la même base (***tic_entreprise***).

Nous en aurons besoin plus loin dans le support :

```
mysql>
CREATE TABLE localite (
id_localite int(5) NOT NULL auto_increment, id_secteur tinyint(3) unsigned NOT NULL, ville
varchar(255) NOT NULL, chiffre_affaires int(10) NOT NULL,   PRIMARY KEY  (id_localite))
ENGINE=InnoDB  DEFAULT CHARSET=latin1;

INSERT INTO localite (id_localite, id_secteur, ville, chiffre_affaires) VALUES
(1, 10, 'paris', 525345), (2, 20, 'marseille', 501236), (3, 30, 'lyon', 377569), (4, 40, 'bordeaux', 350988),
(5, 50, 'paris', 122689);
```

REQUETES DE MODIFICATION

```
UPDATE <Nom de la table>
SET <Nom d'un champ>={valeur}, <Nom d'un champ>={valeur}, . .
WHERE <Critères de sélection>;
```

Langage Naturel (question) :

```
+--------+---------+
| nom    | salaire |
+--------+---------+
| cottet |    1170 |
+--------+---------+
```

- Je souhaite augmenter le salaire de Julien Cottet (employé de l'entreprise) de 1170€ à 1270€.

Langage SQL (requête) :

```
mysql> UPDATE employes SET salaire=1270 WHERE nom='cottet';
```

Résultats (*si nous l'affichons*)

```
+--------+---------+
| nom    | salaire |
+--------+---------+
| cottet |    1270 |
+--------+---------+
```

/! Attention : si deux employés portent le même nom de famille, cela peut poser problème.
Il est donc plus judicieux dans ce cas et même dans n'importe quel cas (pour éviter toute mauvaise surprise) de faire la même requête mais cette fois en se servant de son numéro d'employé pour l'identifié. Ce qui donnerait :

```
mysql> UPDATE employes SET salaire=1270 WHERE id_employes=7369;
```

L'impact sur la table serait le même et c'est la manière la plus judicieuse car le numéro d'employé ne peut être qu'unique (le nom de famille ne l'est pas, il peut arriver que deux personnes puissent porter le même).

Modification d'une table

```
ALTER TABLE  nom_de_la_table
```

Ajout d'une colonne :

```
ALTER TABLE  nom_de_la_table ADD
( nom_de_la_colonne1 type_de_données  [Not Null / Null] ,
  nom_de_la_colonne2 type_de_données  [Not Null / Null] ,
) ;
```

Modification d'une colonne existante :

```
ALTER TABLE  nom_de_la_table MODIFY
( nom_de_la_colonne1 type_de_données  [Not Null / Null] ,
  nom_de_la_colonne2 type_de_données  [Not Null / Null] ,
) ;
```

REQUETES DE REMPLACEMENT

La requête *Replace* est une combinaison d'*insert* et d'*update*, celle-ci fonctionne exactement comme *INSERT*, mais fera l'équivalent d'un *UPDATE* dans le cas où le nombre spécifié pour la clé primaire que l'on tente d'insérér existe déjà dans la table.

Langage Naturel (question) :

- Je souhaite insérer Daniel Chevel à la position 7256 (*id_employes*) car il a changé de service.

Langage SQL (requête) :

```
mysql> INSERT INTO employes (id_employes, prenom, nom, sexe, service, date_embauche, salaire,
        id_secteur) VALUES (7256, 'daniel', 'chevel', 'm', 'marketing', '2010-07-05', 1600, 10);
```

La requête ci-dessus donnera une erreur car la position 7256 est déjà attribuée tandis que la requête ci-dessous permettra de mettre à jour ses informations et ne pas avoir deux fois le même employé (pas de risque de doublon).

```
mysql> REPLACE INTO employes (id_employes, prenom, nom, sexe, service, date_embauche, salaire,
        id_secteur) VALUES (7256, 'daniel', 'chevel', 'm', 'marketing', '2010-07-05', 1600, 10);
```

REQUETES DE SUPPRESSION

```
DELETE FROM <Nom de la table>
WHERE <Critères de sélection>;
```

Langage Naturel (question) :

```
+--------+--------+
| nom    | prenom |
+--------+--------+
| chevel | daniel |
+--------+--------+
```

- Cet employé vient de quitter l'entreprise, il faut donc supprimer son recensement dans notre table.

Langage SQL (requête) :

```
mysql> DELETE FROM employes WHERE nom='chevel';
```

Les requêtes possèdent l'avantage de pouvoir manipuler facilement un grand nombre d'enregistrements sans que l'utilisateur ne doive s'occuper de sélectionner enregistrement par enregistrement. Il lui suffit de spécifier des critères de sélection pour la requête, ainsi que l'opération à effectuer (simple sélection et affichage, insertion, modification ou suppression).

- Supprimer tous les employés (revient à vider la table).

Langage SQL (requête) :

```
mysql> DELETE FROM employes;
```

Dans la finalité, ceci est équivalent à la requête : *truncate*.

AUTRES MOTS-CLES DANS UNE REQUETE

Group By

Nous allons donc maintenant travailler sur notre table « **localite** » qui se trouve sur la base de données « **tic_entreprise** ».

Voici comment elle se présente :

id_localite	id_secteur	Ville	Chiffre (d'affaires)
1	10	Paris	525345
2	20	Marseille	501236
3	30	Lyon	377569
4	40	Bordeaux	350988
5	50	Paris	122689

Nous pouvons y remarquer 2 fois la présence de « Paris ». En effet, si nous nous basons comme une société qui possède une marque de vêtements, il n'est pas impossible que l'on ait plusieurs boutiques de « distribution » dans la même ville.

Langage Naturel (question) :

- Nous souhaitons calculer notre chiffre d'affaires par ville.

Langage SQL (requête) :

```
mysql> SELECT ville, chiffre_affaires FROM localite;
```

Résultats

```
+-----------+------------------+
| ville     | chiffre_affaires |
+-----------+------------------+
| paris     |           525345 |
| marseille |           501236 |
| lyon      |           377569 |
| bordeaux  |           350988 |
| paris     |           122689 |
+-----------+------------------+
```

Problème ! Cette requête sélectionne bien le nom des villes avec leur chiffre d'affaires associé mais nous pouvons y voir la présence de Paris deux fois et nous n'avons pas qu'UN seul chiffre d'affaires pour la ville de Paris. Il faut donc compléter la requête précédente :

Langage SQL (requête) :

```
mysql>  SELECT ville, SUM(chiffre_affaires) FROM localite GROUP BY ville;
```

Résultats

```
+-----------+-----------------------+
| ville     | SUM(chiffre_affaires) |
+-----------+-----------------------+
| bordeaux  |                350988 |
| lyon      |                377569 |
| marseille |                501236 |
| paris     |                648034 |
+-----------+-----------------------+
```

Sum calcule une somme de valeur (*nous l'avons vu dans le chapitre sur les opérateurs arithmétiques*), nous demandons ensuite à mysql de grouper ce chiffre (**Group BY**) en fonction des localités, de cette manière l'addition des deux chiffres d'affaires des deux boutiques parisiennes est bien effectuée.

Having

Sachant que les critères de sélection (WHERE ...) nous permettent d'éliminer un certain nombre d'enregistrements avant la création des groupes, il serait intéressant de disposer d'une deuxième possibilité de filtrage, qui s'applique aux groupes eux-mêmes.

La clause **HAVING** <Critères de sélection des groupes> nous offre la possibilité d'éliminer du résultat les groupes qui ne donnent pas satisfaction aux critères de sélection des groupes.

La requête, ci-dessous, va nous retourner le groupe de boutiques appartenant à une ville qui fait plus de 600 000€ de chiffre d'affaires.

Exemple

```
mysql>  SELECT ville, SUM(chiffre_affaires) FROM localite GROUP BY ville HAVING
SUM(chiffre_affaires)>600000;
```

Résultats

```
+-------+-----------------------+
| ville | SUM(chiffre_affaires) |
+-------+-----------------------+
| paris |                648034 |
+-------+-----------------------+
```

Une des deux boutiques de la ville de Paris n'aurait pas été dans les résultats.

Cependant, à elles deux, elles font plus de 600 000€ de chiffre d'affaires.

Explications

La clause HAVING <Critères de sélection des groupes> est uniquement spécifiée en relation avec un GROUP BY. Une fois les groupes créés, cette clause en élimine certains, basés sur les critères de sélection des groupes.

Les critères de sélection des groupes portent bien entendu sur la valeur d'une ou de plusieurs des fonctions d'agrégation calculées pour chaque groupe.

Syntaxe

```
        SELECT <Liste des champs de groupe>,[COUNT/SUM...]
        FROM <Nom de la table>
        WHERE <Critères de sélection>
GROUP BY <Liste de champs de groupe>
HAVING <Critères de sélection des groupes>
ORDER BY <Nom d'un champ>[ASC/DESC], <Nom d'un champ>[ASC/DESC], …;
```

➢ Les requêtes Sql multi-table

La plupart des bases de données réelles ne sont pas constituées d'une seule table, mais d'un ensemble de tables liées entre elles via certains champs. Par conséquent, les requêtes correspondantes ne sont pas ciblées sur une, mais sur plusieurs tables.

Nous allons différencier trois méthodes pour lier plusieurs tables dans une requête :

- La **jointure**, qui lie plusieurs tables via des champs communs ;
- Les **requêtes imbriquées**, qui utilisent le résultat d'une requête comme source d'une autre.
- Les **unions**

LES UNIONS

Cas d'étude

Sur la base de données « tic_entreprise » nous allons ajouter une table « employes_espagnol » :

```
mysql>
CREATE TABLE IF NOT EXISTS `employes_espagnol` (
  `id_employes` int(3) NOT NULL AUTO_INCREMENT,
  `prenom` varchar(20) NOT NULL,
  `nom` varchar(20) NOT NULL,
  PRIMARY KEY (`id_employes`)
) ENGINE=InnoDB  DEFAULT CHARSET=latin1 AUTO_INCREMENT=4 ;

INSERT INTO `employes_espagnol` (`id_employes`, `prenom`, `nom`) VALUES
(1, 'jose', 'vallas'),
(2, 'alexis', 'sanchez'),
(3, 'lucas', 'carlos');
```

Langage Naturel (question) :

- Je souhaite obtenir les informations sur les employés travaillant dans l'entreprise ET les employés espagnols ?

Langage SQL (requête) :

```
mysql>  SELECT id_employes, nom, prenom FROM employes UNION SELECT id_employes, nom,
        prenom FROM employes_espagnol;
```

UNION est utilisée pour combiner le résultat de plusieurs requêtes SELECT en un seul résultat.

Ainsi, nous obtenons une liste de résultats avec les données contenues dans la table «employes » et les données contenues dans la table « employes_espagnol » dans un seul sous-ensemble de résultat.

Résultats

id_employes	nom	prenom
7256	chevel	daniel
7369	cottet	julien
7499	grand	fabrice
7521	fellier	elodie
7566	lafaye	stephanie
7654	durand	damien
7698	winter	thomas
7782	blanchet	laura
7788	laborde	jean-pierre
7839	desprez	thierry
7844	sennard	emilie
7845	perrin	celine
7846	collier	melanie
7847	perrin	chloe
7848	miller	guillaume
7876	martin	nathalie
7900	lagarde	benoit
7902	vignal	mathieu
7934	thoyer	amandine
9001	vallas	jose
9002	sanchez	alexis
9003	carlos	lucas

Remarques

Si vous n'utilisez pas le mot clef ALL pour l'UNION, toutes les lignes retournées seront uniques, comme si vous aviez fait un DISTINCT pour l'ensemble du résultat.

Si vous spécifiez ALL, vous aurez alors tous les résultats retournés par toutes les commandes SELECT.

Si vous voulez utiliser un ORDER BY pour le résultat final d'UNION, vous devez utiliser des parenthèses.

Vous ne pouvez pas mélanger les clauses UNION ALL et UNION DISTINCT dans la même requête. Si vous utilisez ALL dans une des UNION, alors elle devra être utilisée partout.

Les unions ont un fonctionnement proche des jointures : combiner des informations en provenance de plusieurs sources, en l'occurrence ici, des requêtes.

La différence réside dans le fait que les données combinées doivent être du même type, et les tables doivent avoir les mêmes noms de colonnes.

Cas d'étude

Pour nos exemples suivants nous nous servirons d'une base de données nommée

« **tic_bibliotheque** ».

```
mysql>
CREATE TABLE abonne (
no_abonne int(4) NOT NULL auto_increment,
prenom varchar(15) DEFAULT NULL,
  PRIMARY KEY (no_abonne)
) ENGINE=InnoDB DEFAULT CHARSET=latin1;
INSERT INTO abonne (no_abonne, prenom) VALUES
(1, 'guillaume'),
(2, 'benoit'),
(3, 'chloe'),
(4, 'laura');

CREATE TABLE emprunt (
no_emprunt int(3) NOT NULL auto_increment,
no_livre int(4) default NULL,
no_abonne int(4) default NULL,
date_sortie date default NULL,
date_rendu date default NULL,
PRIMARY KEY  (no_emprunt)
) ENGINE=InnoDB  DEFAULT CHARSET=latin1;
INSERT INTO emprunt (no_emprunt, no_livre, no_abonne, date_sortie, date_rendu) VALUES
(1, 100, 1, '2009-12-17', '2009-12-18'),
(2, 101, 2, '2009-12-18', '2009-12-20'),
(3, 100, 3, '2009-12-19', '2009-12-22'),
(4, 103, 4, '2009-12-19', '2009-12-22'),
(5, 104, 1, '2009-12-19', '2009-12-28'),
(6, 105, 2, '2010-05-20', '2010-06-21'),
(7, 105, 3, '2010-07-05', NULL),
(8, 100, 2, '2010-07-01', NULL);

CREATE TABLE livre (
no_livre int(4) NOT NULL auto_increment,
auteur varchar(25) DEFAULT NULL,
titre varchar(30) DEFAULT NULL,
PRIMARY KEY (no_livre)
) ENGINE=InnoDB DEFAULT CHARSET=latin1;
INSERT INTO livre (no_livre, auteur, titre) VALUES
(100, 'GUY DE MAUPASSANT', 'Une vie'),
(101, 'GUY DE MAUPASSANT', 'Bel-Ami '),
(102, 'HONORE DE BALZAC', 'Le père Goriot'),
(103, 'ALPHONSE DAUDET', 'Le Petit chose'),
(104, 'ALEXANDRE DUMAS', 'La Reine Margot'),
(105, 'ALEXANDRE DUMAS', 'Les Trois Mousquetaires');
```

REQUETE IMBRIQUEE

Langage Naturel (question) :

- Nous aimerions connaître le n° de(s) livre(s) que Chloé n'a pas encore rendu(s) à la bibliothèque

Langage SQL (requête) :

```
mysql> SELECT emprunt.no_livre FROM emprunt WHERE date_rendu IS NULL AND
       emprunt.no_abonne=( SELECT no_abonne FROM abonne WHERE prenom='chloe');
```

Résultats

```
+----------+
| no_livre |
+----------+
|      105 |
+----------+
```

Langage Naturel (question) :

- Nous aimerions maintenant connaître le titre de(s) livre(s) que Chloé n'a pas encore rendu(s) à la bibliothèque.

Langage SQL (requête) :

```
mysql>  SELECT livre.titre FROM livre WHERE livre.no_livre IN
        (SELECT emprunt.no_livre FROM emprunt WHERE emprunt.date_rendu IS NULL AND
        emprunt.no_abonne= (SELECT abonne.no_abonne FROM abonne WHERE
        abonne.prenom='chloe'));
```

Résultats

```
+-------------------------+
| titre                   |
+-------------------------+
| Les Trois Moustequaires |
+-------------------------+
```

Remarques

Une jointure ou requête imbriquée peut mettre en relation plus que 2 tables, c'est le cas ici avec 3 tables.

Différence entre In et égal

- Le signe égal « = » fonctionne avec une seule valeur, une seule ligne de résultat retournée.
- « IN » accepte une série de valeurs et donc plusieurs lignes de résultats.

JOINTURE

Langage Naturel (question) :

- Nous aimerions connaître le nombre de livre(s) emprunté(s) par chaque abonné.

Langage SQL (requête) :

```
mysql>  SELECT a.prenom, COUNT(e.no_livre) AS nb_livre_emprunte
        FROM abonne a, emprunt e
         WHERE a.no_abonne=e.no_abonne
        GROUP BY e.no_abonne;
```

Résultats

```
+-----------+-------------------+
| prenom    | nb_livre_emprunte |
+-----------+-------------------+
| guillaume |                 2 |
| benoit    |                 3 |
| chloe     |                 2 |
| laura     |                 1 |
+-----------+-------------------+
```

Langage Naturel (question) :

- Nous aimerions connaître les dates d'emprunt et de rendu pour l'abonné « guillaume »

Langage SQL (requête) :

```
mysql> SELECT a.prenom, e.date_sortie, e.date_rendu
        FROM abonne a, emprunt e
        WHERE a.no_abonne=e.no_abonne
        AND a.prenom='guillaume';
```

Résultats

```
+-----------+--------------+------------+
| prenom    | date_emprunt | date_rendu |
+-----------+--------------+------------+
| guillaume | 2009-12-17   | 2009-12-18 |
| guillaume | 2009-12-19   | 2009-12-28 |
+-----------+--------------+------------+
```

Remarquez pour l'instant que nous avons préfixé chaque nom d'un champ par le nom de la table correspondante. Au moment où une requête porte sur plusieurs tables nous devons, soit s'assurer que le nom de chaque champ est unique pour l'ensemble des tables, soit adopter la notation <Nom de la table>.<Nom du champ>.

Afin de comprendre le fonctionnement d'une jointure, et surtout celui de la condition de jointure, il est intéressant d'examiner en détail comment SQL procède à l'exécution d'une jointure.

Pour cela, nous allons nous baser sur un exemple de notre base « tic_entreprise ».

Langage Naturel (question) :

- Nous aimerions connaître le nom des villes dans lesquelles travaillent les commerciaux.

Langage SQL (requête) :

```
mysql> SELECT e.prenom, e.nom,l.ville
       FROM employes e, localite l
       WHERE e.service='commercial'
       AND e.id_secteur = l.id_secteur ;
```

Résultats

```
+-----------+---------+-----------+
| prenom    | nom     | ville     |
+-----------+---------+-----------+
| celine    | perrin  | paris     |
| thomas    | winter  | marseille |
| guillaume | miller  | marseille |
| damien    | durand  | lyon      |
| melanie   | collier | lyon      |
| chloe     | perrin  | lyon      |
| emilie    | sennard | bordeaux  |
+-----------+---------+-----------+
```

Comme la clause FROM contient 2 tables, SQL crée d'abord le produit cartésien des deux tables. Pour le produit cartésien, SQL associe à chaque enregistrement de la première table, tous les enregistrements de la deuxième table. Les enregistrements du produit cartésien contiennent donc les champs de la première table suivis des champs de la deuxième table.

Prenom	Nom	Ville
Céline	Perrin	Paris
Thomas	Winter	Marseille
Guillaume	Millet	Marseille
Damien	Durand	Lyon
Mélanie	Collier	Lyon
Chloé	Perrin	Lyon
Emilie	Sennard	Bordeaux

Table employés | Table localité

Une requête imbriquée n'aurait pas pu nous fournir ce résultat car les colonnes font partie de deux tables différentes, dans ce cas, la jointure est la plus appropriée.

En principe, la présence d'une relation (clé étrangère/clé primaire) entre deux tables est une condition nécessaire pour effectuer une jointure sur les tables.
La clause FROM contient les deux tables impliquées dans la jointure.

La clause WHERE contient ce que l'on appelle la condition de jointure. Dans notre exemple, la condition de jointure demande l'égalité des valeurs pour les champs e.id_secteur et l.id_secteur.
La clause SELECT contient les noms des champs à afficher.

Nous pouvons très bien ajouter des opérateurs logiques à notre requête précédente.
Nous allons demander la même chose (le nom des villes dans lesquelles travaillent les commerciaux) en ajoutant la condition suivante : seuls les commerciaux ayant un salaire inférieur à 1600€ doivent apparaître :

```
mysql> SELECT employes.prenom, employes.nom, employes.salaire, localite.ville
       FROM employes, localite where employes.service='commercial' AND employes.id_secteur =
       localite.id_secteur AND employes.salaire<1600 ;
```

La même chose en préfixant les tables :

```
mysql> SELECT e.prenom, e.nom, e.salaire,l.ville FROM employes e, localite l
       WHERE e.service='commercial' AND e.id_secteur = l.id_secteur AND e.salaire<1600 ;
```

Résultats

```
+-----------+---------+---------+-----------+
| prenom    | nom     | salaire | ville     |
+-----------+---------+---------+-----------+
| celine    | perrin  |    1500 | paris     |
| guillaume | miller  |    1500 | marseille |
| damien    | durand  |    1250 | lyon      |
| melanie   | collier |    1500 | lyon      |
| chloe     | perrin  |    1500 | lyon      |
| emilie    | sennard |    1500 | bordeaux  |
+-----------+---------+---------+-----------+
```

Thomas Winter travaillant sur la ville de Marseille n'apparaît plus dans cette liste, bien qu'il soit commercial, il gagne un salaire supérieur à 1600€ (en l'occurrence 1850€).

Remarques

Il existe, en fait, différentes natures de jointures que nous expliciterons plus en détail. Retenez cependant que la plupart des jointures entre tables s'effectuent en imposant l'égalité des valeurs d'une colonne d'une table à une colonne d'une autre table. Nous parlons alors de jointure naturelle.
Nous trouvons aussi des jointures d'une table sur elle-même. Nous parlons alors d'auto-jointure.
Enfin, il arrive que l'on doive procéder à des jointures externes, c'est-à-dire joindre une table à une autre, même si la valeur de liaison est absente dans une table ou l'autre.

JOINTURE INTERNE / EXTERNE

Une jointure entre tables peut être mise en œuvre, soit à l'aide des éléments de syntaxe SQL que nous avons déjà vus (requête imbriquée, jointure), soit à l'aide d'une clause spécifique du SQL, la clause JOIN. Nous allons voir comment à l'aide du SQL, nous pouvons exprimer une jointure avec une syntaxe différente.

Langage Naturel (question) :

- Nous aimerions savoir sur quels secteurs travaillent les employés.(***bdd : tic_entreprise***)

Langage SQL (requête) :

```
mysql> SELECT e.prenom, e.nom, l.ville
       FROM employes e
       INNER JOIN localite l
       ON e.id_secteur = l.id_secteur ;
```

Cette fois, nous récupérons les données depuis une table principale « employes » et nous faisons une jointure interne (INNER JOIN) avec une autre table « localite ».
La liaison entre les champs est faite dans la clause ON sur la dernière ligne.

Langage Naturel (question) :

- Nous aimerions connaître les numéros de livres empruntés en fonction du prénom des abonnés (***bdd: tic_bibliotheque***).

Langage SQL (requête) :

```
mysql> SELECT abonne.prenom, emprunt.no_livre
       FROM abonne
       LEFT JOIN emprunt ON abonne.no_abonne=emprunt.no_abonne
```

Résultats

```
+-----------+----------+
| prenom    | no_livre |
+-----------+----------+
| guillaume |      100 |
| guillaume |      104 |
| benoit    |      101 |
| benoit    |      105 |
| benoit    |      100 |
| chloe     |      100 |
| chloe     |      105 |
| laura     |      103 |
+-----------+----------+
```

Si nous ajoutons un abonné dans la table « abonne », n° 5 au nom de « alex » et que cette personne n'a pas encore emprunté de livre, le résultat généré par la requête ci-dessus sera légèrement différent.

En effet, les jointures externes sélectionnent toutes les données, même si certaines n'ont pas de correspondance dans l'autre table.

La jointure externe est donc plus complète car elle est capable de récupérer plus d'informations, tandis que la jointure interne est plus stricte car elle ne récupère que les données qui ont une équivalence dans l'autre table.

Résultats

```
+-----------+-------------------+
| prenom    | auteur            |
+-----------+-------------------+
| guillaume | GUY DE MAUPASSANT |
| guillaume | ALEXANDRE DUMAS   |
| benoit    | GUY DE MAUPASSANT |
| benoit    | ALEXANDRE DUMAS   |
| benoit    | GUY DE MAUPASSANT |
| chloe     | GUY DE MAUPASSANT |
| chloe     | ALEXANDRE DUMAS   |
| laura     | ALPHONSE DAUDET   |
| alex      | NULL              |
+-----------+-------------------+
```

Langage Naturel (question) :

- Nous aimerions connaître les auteurs des livres empruntés en fonction du prénom des abonnés (***bdd: tic_bibliotheque***).

Langage SQL (requête) :

```
mysql> SELECT abonne.prenom, livre.auteur
       FROM abonne
       LEFT JOIN emprunt ON abonne.no_abonne=emprunt.no_abonne
       LEFT JOIN livre ON emprunt.no_livre=livre.no_livre
```

Résultats

```
+-----------+----------+-------------------+
| prenom    | no_livre | auteur            |
+-----------+----------+-------------------+
| guillaume |      100 | GUY DE MAUPASSANT |
| guillaume |      104 | ALEXANDRE DUMAS   |
| benoit    |      101 | GUY DE MAUPASSANT |
| benoit    |      105 | ALEXANDRE DUMAS   |
| benoit    |      100 | GUY DE MAUPASSANT |
| chloe     |      100 | GUY DE MAUPASSANT |
| chloe     |      105 | ALEXANDRE DUMAS   |
| laura     |      103 | ALPHONSE DAUDET   |
+-----------+----------+-------------------+
```

La clause RIGHT JOIN (jointure externe droite) est également possible.

➢ Les fonctions prédéfinies

DEFINITION

En informatique, une fonction est une portion de code représentant un sous programme, qui effectue une tâche ou un calcul relativement indépendant du reste du programme.

Une fonction contient un ensemble d'instructions réalisant certains traitements.

FONCTIONS D'INFORMATIONS

Indique quelle base de données est sélectionnée :

```
mysql> SELECT DATABASE();
+----------------+
| database()     |
+----------------+
| tic_entreprise |
+----------------+
```

Indique la version de mysql :

```
mysql> SELECT VERSION();
+----------------------+
| VERSION()            |
+----------------------+
| 5.0.51b-community-nt |
+----------------------+
```

FONCTIONS ARITHMETIQUES

Effectue une opération :

```
mysql> SELECT 1+2*3;
+-------+
| 1+2*3 |
+-------+
|     7 |
+-------+
```

Effectue une autre opération (avec calcul en priorité) :

```
mysql> SELECT (1+2)*3;
+---------+
| (1+2)*3 |
+---------+
|       9 |
+---------+
```

Commentaires

```
mysql> SELECT 1+1;     # Ce commentaire se continue jusqu'à la fin de la ligne
mysql> SELECT 1+1;     -- Ce commentaire se continue jusqu'à la fin de la ligne
mysql> SELECT 1; /* Ceci est un commentaire dans la ligne */
```

FONCTIONS TEMPORELLES

Indique la date avec un intervalle de 31 jours :

```
mysql> SELECT DATE_ADD('1998-01-02', INTERVAL 31 DAY);
mysql> SELECT ADDDATE('1998-01-02', 31);
```

```
+----------------------------------------+
| DATE_ADD('1998-01-02', INTERVAL 31 DAY) |
+----------------------------------------+
| 1998-02-02                             |
+----------------------------------------+
```

Indique la date au format Américain (YYYY-MM-DD) :

```
mysql> SELECT CURDATE();
```

```
+------------+
| CURDATE()  |
+------------+
| 2011-05-15 |
+------------+
```

Indique la date au format Américain (YYYYMMDD) numériquement sans les tirets :

```
mysql> SELECT CURDATE() + 0;
```

```
+---------------+
| CURDATE() + 0 |
+---------------+
|      20110515 |
+---------------+
```

Indique l'heure courante au format (HH:MM:SS) :

```
mysql> SELECT CURTIME();
```

```
+-----------+
| CURTIME() |
+-----------+
| 15:55:43  |
+-----------+
```

Indique le nombre de jours entre la date de début et la date de fin :

```
mysql> SELECT DATEDIFF('1997-11-31 23:59:59','1997-12-31');
```

```
+-----------------------------------------------+
| DATEDIFF('1997-11-31 23:59:59','1997-12-31') |
+-----------------------------------------------+
|                                            30  |
+-----------------------------------------------+
```

Indique la date avec une seconde de plus :

```
mysql> SELECT '1997-12-31 23:59:59' + INTERVAL 1 SECOND;
+-----------------------------------------+
| '1997-12-31 23:59:59' + INTERVAL 1 SECOND |
+-----------------------------------------+
| 1998-01-01 00:00:00                     |
+-----------------------------------------+
```

Indique les dates au format Français (***sur la base de données : tic_bibliotheque***) :

```
mysql> SELECT *, DATE_FORMAT(date_rendu,'le %d/%m/%Y a %H:%i:%s') as date_fr FROM emprunt
+-------------+------------+--------------------------+
| date_sortie | date_rendu | date_fr                  |
+-------------+------------+--------------------------+
| 2009-12-17  | 2009-12-18 | le 18/12/2009 a 00:00:00 |
| 2009-12-18  | 2009-12-20 | le 20/12/2009 a 00:00:00 |
| 2009-12-19  | 2009-12-22 | le 22/12/2009 a 00:00:00 |
| 2009-12-19  | 2009-12-22 | le 22/12/2009 a 00:00:00 |
| 2009-12-19  | 2009-12-28 | le 28/12/2009 a 00:00:00 |
| 2010-05-20  | 2010-06-21 | le 21/06/2010 a 00:00:00 |
| 2010-07-05  | NULL       | NULL                     |
| 2010-07-01  | NULL       | NULL                     |
+-------------+------------+--------------------------+
```

Indique le jour d'une date en particulier :

```
mysql> SELECT DAYNAME('2011-03-28');
+-----------------------+
| DAYNAME('2011-03-28') |
+-----------------------+
| Monday                |
+-----------------------+
```

Indique le jour à la date d'aujourd'hui :

```
mysql> SELECT DAYNAME(CURDATE());
+--------------------+
| DAYNAME(CURDATE()) |
+--------------------+
| Sunday             |
+--------------------+
```

Indique le numéro du jour de l'année :

```
mysql> SELECT DAYOFYEAR('1998-02-05');
+-------------------------+
| DAYOFYEAR('1998-02-05') |
+-------------------------+
|                      36 |
+-------------------------+
```

Indique la date courante :

```
mysql> SELECT NOW();
+---------------------+
| NOW()               |
+---------------------+
| 2011-05-15 16:10:51 |
+---------------------+
```

FONCTIONS DE CHAINES DE CARACTERES

Permet de crypter un mot de passe :

```
mysql> SELECT PASSWORD('mypass') ;
+-------------------------------------------+
| PASSWORD('mypass')                        |
+-------------------------------------------+
| *6C8989366EAF75BB670AD8EA7A7FC1176A95CEF4 |
+-------------------------------------------+
```

La fonction CONCAT_WS() signifie CONCAT With Separator, c'est-à-dire "concaténation avec séparateur :

```
mysql> SELECT CONCAT_WS("-","Premier nom","Deuxième nom","Dernier nom");
+-----------------------------------------------------------+
| CONCAT_WS("-","Premier nom","Deuxième nom","Dernier nom") |
+-----------------------------------------------------------+
| Premier nom-Deuxième nom-Dernier nom                      |
+-----------------------------------------------------------+
```

Indique le nombre de caractères dans une chaîne :

```
mysql> SELECT LENGTH('moi');
+---------------+
| LENGTH('moi') |
+---------------+
|             3 |
+---------------+
```

Indique la position d'un caractère dans une chaîne :

```
mysql> SELECT LOCATE('j', 'aujourdhui');
+---------------------------+
| LOCATE('j', 'aujourdhui') |
+---------------------------+
|                         3 |
+---------------------------+
```

Recherche une chaine de caractère dans une colonne (***sur la base de données : tic_entreprise***) avec index en FULLTEXT :

```
mysql> SELECT prenom,nom FROM employes WHERE MATCH (service) AGAINST ('informatique');
+---------+--------+
| prenom  | nom    |
+---------+--------+
| daniel  | chevel |
| mathieu | vignal |
+---------+--------+
```

Remplace les caractères choisis par d'autres :

```
mysql> SELECT REPLACE('www.creajoy.com', 'w', 'W');
+-------------------------------------+
| REPLACE('www.creajoy.com', 'w', 'W') |
+-------------------------------------+
| WWW.creajoy.com                      |
+-------------------------------------+
```

Coupe une chaîne de caractères :

```
mysql> SELECT SUBSTRING('bonjour',4);
+------------------------+
| SUBSTRING('bonjour',4) |
+------------------------+
| jour                   |
+------------------------+
```

Supprime les caractères d'espacement en début et en fin de chaîne :

```
mysql> SELECT TRIM('  bonsoir   ');
+----------------------+
| TRIM('  bonsoir   ') |
+----------------------+
| bonsoir              |
+----------------------+
```

Transforme le texte en majuscules :

```
mysql> SELECT UPPER('Hey');
+--------------+
| UPPER('Hey') |
+--------------+
| HEY          |
+--------------+
```

➢ Tables Virtuelles

INTRODUCTION

Le langage SQL (nous le rappellons : acronyme de Structured Query Language - Langage Structuré de Requêtes), a été conçu pour gérer les données dans un SGBDR. A l'aide des DML (Data Manipulation Language les requêtes *SELECT, INSERT, UPDATE, DELETE*) il est possible de manipuler ces données qui sont stockées dans des tables.

SQL nous propose une autre interface pour accéder à cette information: ***les vues.***

Nous allons voir comment créer et se servir des vues, puis avec quelques exemples pratiques, nous allons voir comment les utiliser le mieux possible.

QU'EST-CE QU'UNE VUE ?

Les vues sont des tables virtuelles issues de l'assemblage d'autres tables en fonction de critères. Techniquement, les vues sont créées à l'aide d'une requête *SELECT*. Elles ne stockent pas les données qu'elles contiennent mais conservent juste la requête permettant de les créer.

La requête *SELECT* qui génère la vue référence d'une ou plusieurs tables. La vue peut donc être, par exemple, une jointure entre différentes tables, l'agrégation ou l'extraction de certaines colonnes d'une table. Elle peut également être créée à partir d'une autre vue.

A QUOI SERVENT LES VUES ?

Les vues peuvent être utilisées pour différentes raisons. Elles permettent de :

- Contrôler l'intégrité en restreignant l'accès aux données pour améliorer la confidentialité.
- Partitionnement vertical et/ou horizontal pour cacher des champs aux utilisateurs, ce qui permet de personnaliser l'affichage des informations suivant le type d'utilisateur.
- Masquer la complexité du schéma.
- Indépendance logique des données, utile pour donner aux utilisateurs l'accès à un ensemble de relations représentées sous la forme d'une table. Les données de la vue sont alors des champs de différentes tables regroupées, ou des résultats d'opérations sur ces champs.
- Modifier automatiquement des données sélectionnées (sum(), avg(), max(),...).
- Manipuler des valeurs calculées à partir d'autres valeurs du schéma.
- Conserver la structure d'une table si elle doit être modifiée.
- Le schéma peut ainsi être modifié sans qu'il ne soit nécessaire de changer les requêtes du côté applicatif.

CREER UNE VUE

Syntaxe d'une vue

```
mysql> CREATE VIEW
```

La commande MySQL pour créer une vue est assez proche de la syntaxe du standard SQL.

```
mysql> CREATE VIEW nom_de_la_vue AS requête_select
```

Exemple (*sur notre table « employes » de notre bdd :* ***tic_entreprise***) :

```
mysql> CREATE VIEW vue_homme (prenom, nom, sexe, service) AS SELECT prenom, nom, sexe,
service FROM employes WHERE sexe='m';
```

Cette vue va nous permettre de retenir certaines informations sur la table employés et seulement sur les hommes de l'entreprise.

UTILISER UNE VUE

Appel de la vue

```
mysql> SELECT prenom FROM vue_homme;
```

Résultats

```
+-------------+
| prenom      |
+-------------+
| daniel      |
| julien      |
| fabrice     |
| damien      |
| thomas      |
| jean-pierre |
| thierry     |
| guillaume   |
| benoit      |
| mathieu     |
+-------------+
```

Cette vue ne nous retourne bien que les prénoms des hommes dans l'entreprise.

Les vues sont stockées dans la table « **views** » de la base de données « **information_schema** ».

Il est possible de les voir en faisant les requêtes suivantes:

```
mysql> USE information_schema ;
```

Puis :

```
mysql> SELECT * FROM views;
```

SUPPRIMER UNE VUE

Il est possible de supprimer une vue (si l'on se trouve sur la base de données qui est concernée par la vue) en faisant la requête suivante :

```
mysql> DROP VIEW  vue_homme ;
```

Une fois la vue créée, il est bien évidemment possible de la modifier avec la commande ALTER VIEW. Il existe encore d'autres commandes plus approfondies dans le contexte de la création de vues.

➢ Tables Temporaires

INTRODUCTION

L'utilisation d'une table temporaire ne dépasse jamais la session SQL en cours, il peut être utile de l'utiliser pour créer un sous-ensemble d'une autre table plus léger afin de soulager le serveur SQL.

Travailler avec des tables SQL de 30 Mo pour filtrer et récupérer les données peut être une épreuve lourde pour votre serveur.
La fonction CREATE TEMPORARY TABLE peut vous aider à alléger grandement les performances.

EXEMPLE

Exemple sur la base de données (***tic_entreprise***) :

```
mysql> CREATE TEMPORARY TABLE temp SELECT * FROM employes WHERE sexe='f';
```

Utilisation

```
mysql> SELECT * FROM temp;
```

Si la session SQL est expirée, cette requête ne fonctionnera plus.

DIFFERENCES TABLES TEMPORAIRES ET TABLES VIRTUELLES

- Dans la table temporaire nous sauvegardons les données, si nous changeons le prénom d'un employé dans la table « employes », il ne changera pas dans la table temporaire précédemment créée car nous ne touchons pas aux mêmes données.

- Dans la table virtuelle (vue) nous sauvegardons la requête qui permet de mener aux données, si nous changeons le prénom d'un employé dans la vue, il change en conséquence dans la table et INVERSEMENT. (nous touchons aux mêmes données).

➢ Transactions

DEFINITION

En informatique, et particulièrement dans les bases de données, une transaction telle qu'une réservation, un achat ou un paiement est mise en œuvre via une suite d'opérations qui font passer la base de données d'un état A - antérieur à la transaction - à un état B postérieur.
Le concept de transaction s'appuie sur la notion de point de synchronisation (*sync point*) qui représente un état stable du système informatique considéré en particulier sur ces données.

Habituellement, lorsqu'une commande permettant de modifier les informations dans une table (INSERT, UPDATE, DELETE), il n'est plus possible de faire marche arrière.
Nous ne sommes jamais à l'abri d'une erreur, C'est pour cela qu'InnoDB intègre une notion d'annulation.
En résumé, vous pouvez faire un retour en arrière dans votre base de données.

Exemple de transaction dans le monde bancaire

Par exemple lors d'une opération informatique de transfert d'argent d'un compte bancaire sur un autre compte bancaire, il y a une tâche de retrait d'argent sur le compte source et une de dépôt sur le compte cible. Le programme informatique qui effectue cette transaction va s'assurer que les deux opérations peuvent être effectuées sans erreur, et dans ce cas, la modification deviendra alors effective sur les deux comptes. Si ce n'est pas le cas l'opération est annulée. Les deux comptes gardent leurs valeurs initiales. Nous garantissons ainsi la cohérence des données entre les deux comptes.

UTILISATION

Démarre la zone de la mise en tampon

```
mysql> START TRANSACTION;
```

Requête de modification :

```
mysql> UPDATE employes SET prenom='erreur'  WHERE  id_employes=7256;
```

Affiche le contenu de la table :

```
mysql> SELECT * FROM employes;
```

Donne l'ordre à MySQL de tout annuler depuis START TRANSACTION :

```
mysql> ROLLBACK;
```

Au contraire, COMMIT, valide l'ensemble de la transaction :

```
mysql> COMMIT;
```

Si, ni de ROLLBACK, ni de commit n'est spécifié, mysql annulera l'opération à la déconnexion et effectuera donc un ROLLBACK.

POINT DE RESTAURATION

Il est également possible de créer des points de restauration grâce aux « SAVEPOINT ».

```
mysql> START TRANSACTION;
mysql> SELECT * FROM employes;
mysql> SAVEPOINT point1;
mysql> UPDATE employes SET prenom='Julien A' WHERE id_employes=7369;

mysql> SELECT * FROM employes;
mysql> SAVEPOINT point2;
mysql> UPDATE employes SET prenom='Julien B' WHERE id_employes=7369;

mysql> SELECT * FROM employes;
mysql> SAVEPOINT point3;
mysql> ROLLBACK TO point2;

mysql> SELECT * FROM employes;
```

Les SAVEPOINT permettent de revenir ultérieurement à un point antérieur sans faire un ROLLBACK sur la totalité de la transaction.
Dans ce cas, nous revenons au prénom : Julien A.
Lorsque l'on effectue un ROLLBACK sur le point2, nous ne pourrons pas revenir sur le point3 mais uniquement les points antérieurs (au point2) comme le point1.

➢ Requête préparées

INTRODUCTION

Les requêtes préparées (prepared statements) permettent d'automatiser une requête qui apparaît souvent et ainsi alléger le temps de traitement.

La requête ne doit être analysée (ou préparée) qu'une seule fois, mais peut être exécutée plusieurs fois avec des paramètres identiques ou différents. Lorsque la requête est préparée, la base de données va analyser, compiler et optimiser son plan pour exécuter la requête. Pour les requêtes complexes, ce processus peut prendre assez de temps, ce qui peut ralentir vos applications si vous devez répéter la même requête plusieurs fois avec différents paramètres.
En utilisant les requêtes préparées, vous évitez ainsi de répéter le cycle analyser/compilation/optimisation.
Pour résumer, les requêtes préparées utilisent moins de ressources et s'exécutent plus rapidement.

DECLARATION

Exemple sur la base de données (tic_bibliotheque) :

```
PREPARE test FROM 'SELECT * FROM abonne WHERE prenom=?';
```

Nous déclarons une requête préparée qui se nomme « test » et qui contient un point d'interrogation « ? », ceci est un marqueur destiné à recevoir un argument.

Affectation d'une valeur à une variable de session

```
SET @mavariable='laura';
```

UTILISATION

```
EXECUTE test USING @mavariable ;
```

La commande « Execute » permet d'exécuter la requête préparée en utilisant une variable afin de renseigner le marqueur du point d'interrogation.

SUPPRESSION

```
DROP PREPARE test;
```

DUREE DE VIE

La requête existe dans la session où elle a été déclarée, dans une autre session elle ne sera pas accessible.

Attention à ne pas nommer une nouvelle requête préparée avec le nom d'une requête préparée déjà déclarée.

➢ Procédures Stockées

DEFINITION

Une procédure stockée (ou stored procedure en anglais) est un ensemble d'instructions SQL précompilées, stockées dans une base de données et exécutées sur demande par le SGBD qui manipule la base de données.
Les procédures stockées servent donc à appeler des requêtes directement stockées sur le serveur.

FONCTIONNEMENT

Les requêtes envoyées à un serveur SQL font l'objet d'une 'analyse syntaxique' puis d'une interprétation avant d'être exécutées. Ces étapes peuvent être très lourdes dès lors que nous envoyons plusieurs requêtes complexes.

Les procédures stockées répondent à ce problème : une requête n'est envoyée qu'une unique fois sur le réseau puis analysée, interprétée et stockée sur le serveur sous forme exécutable (précompilée). Pour qu'elle soit exécutée, le client n'a qu'à envoyer une requête comportant le nom de la procédure stockée.
On peut ainsi passer des paramètres à une procédure stockée lors de son appel, et recevoir le résultat de ses opérations comme celui de toute requête SQL.

INTERET

L'intérêt est multiple :

- Diminution du temps d'exécution - Théoriquement une requête stockée est plus rapide qu'une requête « envoyé » car elle n'est pas retraduite à chaque exécution. Cela peut également être lancé de façon automatique par un événement déclencheur (de l'anglais "trigger").
- simplification : code plus simple à comprendre
- rapidité : moins d'informations sont échangées entre le serveur et le client
- performance : économise au serveur l'interprétation de la requête car elle est précompilée
- sécurité : les applications et les utilisateurs n'ont aucun accès direct aux tables, mais passent par des procédures stockées prédéfinies

Informations

Nous allons changer le délimiter « ; » en « | ».

```
DELIMITER |
```

Ainsi, si mysql croise un « ; » dans le code des procédures stockées, il ne pensera pas que c'est la fin du code mais bien une instruction au même titre que les autres.

DÉCLARATION

Procédure « test » :

```
CREATE PROCEDURE test(IN valeur VARCHAR(10))
BEGIN
SELECT valeur;
END|
```

Dans les parenthèses apparaissent les paramètres (sens_du_parametre nom_parametre type_parametre).

IN : précise donc que la valeur entre la parenthèse sera entrante, il s'agit du sens.

OUT : le paramètre sera une variable de sortie afin d'y conserver un résultat.

INOUT : indiquera que le paramètre sera une variable d'entrée-sortie, la procédure connaît sa valeur en entrée et peut la renvoyer avec une valeur modifiée.

Begin et End définissent respectivement le début et la fin d'un bloc d'instructions.

UTILISATION

```
CALL  test(allo)|
```

Cette ligne permet d'exécuter la procédure « test ».

OBSERVATION

```
SHOW PROCEDURE STATUS \G|
```

Le « \G » n'est pas obligatoire mais participe à obtenir un affichage plus lisible en mettant les informations en lignes plutôt qu'à la suite.

Autre exemple

Avec une variable de sortie (out) et sur la base de données (tic_entreprise) :

```
CREATE PROCEDURE compte(OUT result INT)
BEGIN
SELECT count(*) from employes INTO result;
END|
call compte(@r)|
select @r|
```

CONDITIONS

```
CREATE PROCEDURE condition_couleur(IN val VARCHAR(10))
BEGIN
	IF val = 'bleu' THEN
SELECT 'il s\'agit du bleu';
ELSEIF val = 'rouge' THEN
		SELECT 'il s\'agit du rouge';
	ELSE
		SELECT 'autre couleur';
	END IF;
END|
CALL condition_couleur('bleu')|
```

ARGUMENTS (PARAMETRES)

Les arguments (parameters) ont un sens dans les procédures stockées.

- IN:
  - Paramètre par défaut;
  - Valeur passée à la procédure,
  - peut-être modifiée dans la procédure
  - mais reste inchangé à l'extérieur de l'appel.
- OUT:
  - Aucune valeur passée à la procédure,
  - peut être modifiée dans la procédure,
  - et la valeur est changée à l'extérieur de la procédure.
- INOUT:
  - Caractéristiques des paramètres IN et OUT:
  - Une valeur est fournie à la procédure,
  - et peut être modifiée et retournée.

SPECIFICITES

Déterministe :

Définition du déterminisme (informatique) : Traitement qui donne un même résultat à mêmes conditions d'entrée et code d'exécution. Par opposition toutes les fonctions NON DÉTERMINISTES ne donnent pas les mêmes résultats si même exécution, comme RAND(), NEWID(), CURRENT_TIMESTAMP, GETDATE()… mais aussi celles que l'on peut créer (UDF). En effet par définition une fonction ou expression qui ne donne jamais le même résultat ne peut être persistante et la colonne n'est pas indexable.

Citation

Lorsque l'on demande l'heure à Woody Allen, il répond systématiquement « je ne peux pas vous la donner, ça change tout le temps »… C'est une bonne façon de comprendre l'indéterminisme.

Type de sécurité

Par défaut (cf SQL SECURITY à DEFINER) les utilisateurs qui ont accès à une base de données associée ont le droit d'exécuter les routines déclarées sur cette base.

Accès aux données SQL:

- Contains SQL : renseigne sur le fait que la procédure interagit avec la base.
- No SQL : indique l'inverse et donc que la procédure n'interagit pas avec la base.
- Reads SQL DATA : précise que les interactions sont en lecture seulement.
- Modifies SQL DATA : signifie que les mises à jour de la base sont possibles.

DIFFÉRENCES : PROCÉDURE STOCKÉE ET REQUÊTE PRÉPARÉE

Une requête préparée est disponible uniquement dans la session en cours tandis que la procédure stockée est accessible indéfiniment tant qu'on ne la supprime pas.
Contrairement à la requête préparée, la procédure stockée peut avoir un ensemble d'instructions permettant de réaliser un certain nombre d'actions bien précis.
Les procédures stockées représentent un bon moyen d'optimiser les requêtes redondantes, tout en offrant des possibilités suffisamment larges pour permettre des actions variées.

➢ Structures itératives (Boucle)

DEFINITION

Nous appelons structure itérative (ou récursive), la structure qui permet d'exécuter plusieurs fois les mêmes instructions. Elle permet de faire "en boucle" un bloc d'instructions.

EXEMPLE

WHILE

```
CREATE PROCEDURE dowhile()
BEGIN
 DECLARE v1 INT DEFAULT 5;

 WHILE v1 > 0 DO
  ...
  SET v1 = v1 - 1;
 END WHILE;
END
```

LOOP / ITERATE / LEAVE

```
CREATE PROCEDURE doiterate(p1 INT)
BEGIN
 label1: LOOP
  SET p1 = p1 + 1;
  IF p1 < 10 THEN
   ITERATE label1;
  END IF;
  LEAVE label1;
 END LOOP label1;
 SET @x = p1;
END;
```

REPEAT

```
CREATE PROCEDURE dorepeat(p1 INT)
   BEGIN
    SET @x = 0;
    REPEAT
     SET @x = @x + 1;
   UNTIL @x > p1 END REPEAT;
  END
```

➢ Structures Conditionnelles

DEFINITION

Nous appelons structure conditionnelle, la structure qui permet de réaliser une (ou plusieurs) instruction(s) sous certaines conditions. Elle nécessite l'utilisation des opérateurs de comparaisons et parfois des opérateurs logiques.

EXEMPLE

Dans une procédure, affichage du salaire d'un employé et son groupe en fonction de son prénom :

```
CREATE PROCEDURE groupe(in entree_p varchar(10))
BEGIN
	DECLARE s INT DEFAULT 0;
	SELECT * FROM employes WHERE prenom=entree_p;
	SELECT salaire FROM employes WHERE prenom = entree_p INTO s;
	IF s > 2000  THEN
		SELECT 'le salarie fait partie du Groupe 1';
	ELSEIF s < 3000 THEN
		SELECT 'le salarie fait partie du Groupe 3';
	ELSE
		SELECT 'le salarie fait partie du Groupe 2';
	END IF;
END|
CALL groupe('julien')|
```

Ceci peut aussi s'effectuer avec l'instruction : case.

➢ Curseur

DEFINITION

Les curseurs intégrés dans des procédures stockées permettent une meilleure lecture des résultats lorsqu'une requête SELECT renvoie plus d'une ligne. Cela permet également de faire des traitements sur les résultats, cependant dans Mysql les curseurs sont en lecture seule et non navigable.

EXEMPLE

```
CREATE PROCEDURE curdemo()
BEGIN
  DECLARE done INT DEFAULT 0;
  DECLARE a CHAR(16);
  DECLARE b,c INT;
  DECLARE cur1 CURSOR FOR SELECT id,data FROM test.t1;
  DECLARE cur2 CURSOR FOR SELECT i FROM test.t2;
  DECLARE CONTINUE HANDLER FOR SQLSTATE '02000' SET done = 1;

  OPEN cur1;
  OPEN cur2;

  REPEAT
    FETCH cur1 INTO a, b;
    FETCH cur2 INTO c;
    IF NOT done THEN
      IF b < c THEN
        INSERT INTO test.t3 VALUES (a,b);
      ELSE
        INSERT INTO test.t3 VALUES (a,c);
      END IF;
    END IF;
  UNTIL done END REPEAT;

  CLOSE cur1;
  CLOSE cur2;
END
```

➢ Déclencheurs (triggers)

INTRODUCTION

Dans les bases de données, lors de la mise à jour ou de la suppression d'une donnée, si un déclencheur existe, il peut lancer automatiquement une procédure stockée, qui agit en parallèle sur la même donnée dans une table afférente. Cela permet d'automatiser certains traitements assurant la cohérence et l'intégrité de la base de données.

Les déclencheurs peuvent être définis pour s'exécuter avant ou après une commande INSERT, UPDATE ou DELETE, soit une fois par ligne modifiée, soit une fois par expression SQL.

CRÉATION

```
CREATE TRIGGER employes2 AFTER INSERT ON employes
FOR EACH ROW
BEGIN
UPDATE employes_informations SET nombre = nombre + 1, dernier = new.id_employes,
derniere_embauche = new.date_embauche;
 INSERT INTO employes_sauvegarde (id_employes, prenom, nom, sexe, service, date_embauche,
salaire, id_secteur) VALUES (new.id_employes, new.prenom, new.nom, new.sexe, new.service,
new.date_embauche, new.salaire, new.id_secteur);
END|
```

FONCTIONNEMENT

Dans cet exemple, nous avons créé une table « employes_sauvegarde » à l'identique de la table « employes » sur la base de données tic_entreprise.
Ce trigger se déclenchera seul lors d'un événement, en l'occurrence ici : à chaque fois qu'un employé sera inséré sur la table « employes », il le sera aussi sur la table « employes_sauvegarde ».

Les triggers qui contiennent les mots-clés NEW et OLD font référence à la ligne qui est insérée (NEW), supprimée (OLD) ou alors mise à jour (OLD = ancienne version, NEW = nouvelle version) sur la table sur laquelle nous avons créé le trigger.

En effet un trigger se déclenche seul après une action particulière, il convient donc de préciser les informations suivantes :

- Table : Nom de la table concernée par le trigger
- Moment : Avant ou après l'action entraînant le déclenchement
- Evénements : Lors d'un Insert, Update ou Delete.

OBSERVATION

```
SHOW TRIGGERS \G|
```

Le « \G » n'est pas obligatoire mais participe à obtenir un affichage plus lisible en mettant les informations en lignes plutôt qu'à la suite.

SUPPRESSION

```
DROP TRIGGER exemple1|
```

Cette ligne permet de supprimer un trigger afin qu'il ne se déclenche plus lors d'un événement particulier.
Sa suppression n'entraîne pas la suppression des actions réalisées lors de ses déclenchements passés.

➢ Evénements

DEFINITION

Les événements (*Event Scheduler)* permettent de déclencher des requêtes SQL ou encore des procédures stockées à des moments précis dans le temps. Pour que cela fonctionne, il ne faut pas oublier d'activer le processus d'événement sous Mysql.
Grâce à un déclencheur temporel, l'évènement peut réaliser une requête SQL ou une combinaison de requête via une procédure stockée de une à plusieurs requêtes SQL programmées par l'utilisateur. Le déclenchement peut être périodique et se lancer de une à plusieurs fois.

Les événements n'ont pas été introduits pour rien dans Mysql car ils permettent d'automatiser un bon nombre de tâches, son utilisation est donc intéressante ; voici quelques exemples :

- Programmer des requêtes de suppression pour délester de vieilles discussions pour votre chat.
- Calcul annuel des salaires, primes, charges, bilan, etc.
- Programmer des requêtes de sauvegardes automatiques chaque nuit.
- Différer l'exécution d'un traitement lourd en ressources à un autre moment.
- Analyser et optimiser l'ensemble des tables mises à jour dans la journée

ETAT ET TYPES

Etat : Activation / Désactivation

Permet de savoir si les événements sont actifs ou inactifs :

```
SHOW GLOBAL VARIABLES LIKE 'event_scheduler';
```

```
+-----------------+-------+
| Variable_name   | Value |
+-----------------+-------+
| event_scheduler | OFF   |
+-----------------+-------+
```

Nous affectons la valeur « 1 » dans la variable concernée et cela permet d'activer les événements sous Mysql.

```
SET GLOBAL event_scheduler = 1 ;
```

Il est possible d'activer les événements de manière permanente en éditant le fichier my.ini de mysql.

- Enabled 1 : signifie que le service est volontairement coupé et ne pourra pas être activé sans un redémarrage et une modification des paramètres de configuration.
- Disabled 0 : signifie que le service est actif.
- SLAVESIDE_DISABLED 2 : signifie que le service est suspendu. Néanmoins une requête SQL peut à tout moment le réactiver.

Type d'événements

- One time : Une fois
- Reccuring : Récurrent

EXEMPLE

Cet événement permet d'insérer une ligne toutes les minutes dans la table.

```
CREATE EVENT EXEMPLE_1
        ON SCHEDULE  EVERY 1 MINUTE
                DO INSERT INTO nomdelabdd.nomdelatable (champ) VALUES ('Exemple 1');
```

Le mot clef EVERY indique que l'évènement est récurrent. Il est suivi par l'intervalle entre chaque répétition.

Lancer (une seule fois) une PROCÉDURE STOCKÉE à 03h00 du 1er janvier 2020

```
CREATE EVENT EXEMPLE_2
        ON SCHEDULE AT '2020-01-01 03:00:00'
                DO CALL nomdelaprocedure('argument');
```

Attention : Cet exemple n'a pas pris en compte le décalage horaire avec GMT.

Requêtes utiles liées à l'utilisation des événements :

Renommer un événement :

```
ALTER EVENT EXEMPLE_1 RENAME TO NOUVEAU_EXEMPLE_1;
```

Modifiez la fréquence d'un évent existant :

```
ALTER EVENT EXEMPLE_1 ON SCHEDULE EVERY 10 MINUTE;
```

Suppression d'un évent :

```
DROP EVENT IF EXISTS EXEMPLE_1;
```

Désactiver un évent :

```
ALTER EVENT EXEMPLE_1 DISABLE;
```

Activer un évent (sous réserve que le gestionnaire d'événements soit activé) :

```
ALTER EVENT EXEMPLE_1 ENABLE;
```

Contrôler l'état d'un évent :

```
SELECT EVENT_NAME, STATUS FROM INFORMATION_SCHEMA.EVENTS; show events;
```

DIFFERENCE TRIGGER ET EVENT

Un Trigger (déclencheur) se déclenche suite à une action de l'utilisateur tel : insert, update, delete.

Un évent (événement) se déclenche à un instant T dans le temps peu importe les actions de l'utilisateur.

➢ Contraintes d'intégrités

DEFINITION

Dans une base de données, une contrainte d'intégrité permet de garantir la cohérence des données lors des mises à jour de la base. En effet, les données d'une base ne sont pas indépendantes, mais obéissant à des règles sémantiques, après chaque mise à jour, le SGBD contrôle qu'aucune contrainte d'intégrité n'est violée.

L´intégrité référentielle est une situation dans laquelle pour chaque information d'une table A qui fait référence à une information d'une table B, l'information référencée existe dans la table B. L'intégrité référentielle est un vecteur de cohérence du contenu de la base de données.

Par exemple : on définira qu'un livre a un ou plusieurs auteurs. Une contrainte d'intégrité référentielle interdira l'effacement d'un auteur, tant que dans la base de données il existera au moins un livre se référant à cet auteur. Cette contrainte interdira également d'ajouter un livre si l'auteur n'est pas préalablement inscrit dans la base de données.

LES REFERENCES

Voici deux tables (en innodb) :

articles
- id_article
- titre
- couleur
- prix
- id_fourn

fournisseur
-id_fournisseur
- nom
- ville

Pour ajouter une contrainte sur un champ, il faut lui ajouter un index (onglet structure de PMA).
Dans ce même onglet, il y a un lien « gestion des relations ».
Nous allons lier en référence le champ « id_fournisseur » de la table fournisseur au champ « id_fourn » de la table articles. Ce qui donnera : « id_fourn->id_fournisseur ».

De cette manière je ne pourrais ajouter d'articles si aucun fournisseur n'existe.
Chaque article sera forcément rattaché à un fournisseur existant.

GESTION DES RELATIONS

Le champ « id_fournisseur » de la table fournisseur est une référence du champ « id_fourn » de la table articles. Il peut y avoir à présent plusieurs types de relations possibles :

- on delete CASCADE :
  si nous supprimons un fournisseur, les articles qui lui sont rattachés seront également supprimés.

- on delete SET NULL :
  si nous supprimons un fournisseur, les articles qui lui sont rattachés auront la valeur NULL (attention il faut auparavant que le champ « id_fourn » accepte la valeur NULL par défaut).

- on delete NO ACTION :
  si nous souhaitons supprimer un fournisseur et que des articles lui sont rattachés, alors la requête DELETE est exécutée mais restaurée aussitôt. Autrement dit: tant que des articles sont rattachés à ce fournisseur nous ne pourrons pas le supprimer.

- on delete RESTRICT :
  si nous souhaitons supprimer un fournisseur et que des articles lui sont rattachés, le comportement sera le même que pour no action mais l'exécution ne se produit pas du tout.

Ces actions sont valables en cas de suppression mais aussi en cas de modification.

- on update CASCADE :
  si nous modifions un fournisseur, les articles qui lui sont rattachés seront également modifiés

- on update SET NULL :
  Si nous modifions un fournisseur (par exemple son id_fournisseur), les articles qui lui sont rattachés auront la valeur NULL même si le fournisseur existe toujours sous un autre numéro d'id. (Attention : il faut auparavant que le champ accepte la valeur NULL par défaut).

- on update NO ACTION :
  Si nous souhaitons modifier un fournisseur et que des articles lui sont rattachés, alors la requête UPDATE est exécutée mais restaurée aussitôt. Autrement dit: tant que nous avons des articles rattachés à ce fournisseur nous ne pourrons pas le modifier.

- on update RESTRICT :
  si nous souhaitons supprimer un fournisseur et que des articles lui sont rattachés, le comportement sera le même que pour no action mais l'exécution ne se produit pas du tout.

➢ PHPMYADMIN

INTRODUCTION

Nous allons maintenant faire des manipulations sur une base de données. Nous allons voir ce que peuvent contenir une base et ses tables.

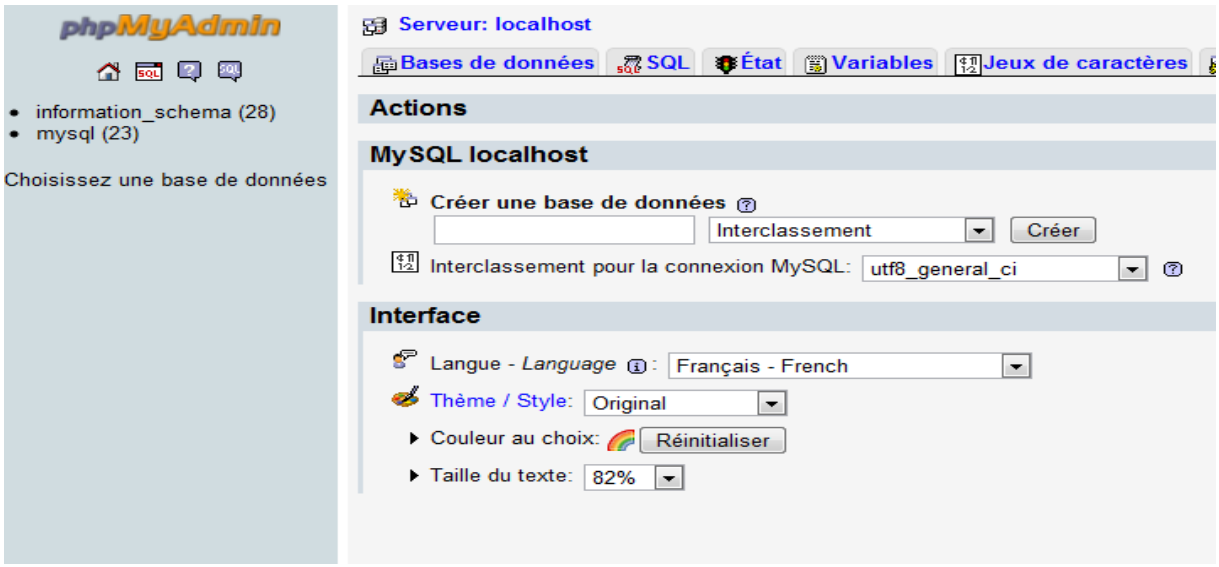
Il existe plusieurs façons d'accéder à sa base de données et d'y faire des modifications. Nous pouvons utiliser une ligne de commande (console), exécuter les requêtes en PHP ou faire appel à un programme qui nous permet d'avoir rapidement une vue d'ensemble. Nous allons donc découvrir l'interface de PhpMyAdmin, un des outils les plus connus permettant de manipuler une base de données MySQL.

PhpMyAdmin est livré avec WAMP, vous allez donc pouvoir l'utiliser tout de suite. La quasi-totalité des hébergeurs permettent d'utiliser PhpMyAdmin. Renseignez-vous auprès de votre hébergeur pour savoir comment y accéder. Vous aurez très certainement besoin d'un login et d'un mot de passe.

Il y a plusieurs façons d'accéder à l'interface de PhpMyAdmin, vous pouvez par exemple le faire via la page d'accueil de wamp.

INTERFACE

L'accueil de phpMyAdmin ressemble à ceci :



La partie en couleur « turquoise » représente la liste de vos bases de données.

Le nombre entre parenthèses correspond au nombre de tables qu'il y a dans chaque base.

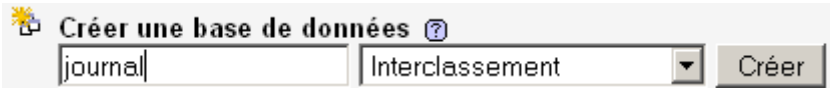
Sur la capture d'écran, nous avons donc 2 bases de données : information_schema, qui contient 28 tables, et mysql, qui contient 23 tables.

CREER UNE BASE DE DONNEES

Pour créer une nouvelle base de données, tapez un nom dans le champ de formulaire à droite, cliquez sur "Créer". (Cela correspond à la commande : *create database nom ;)*

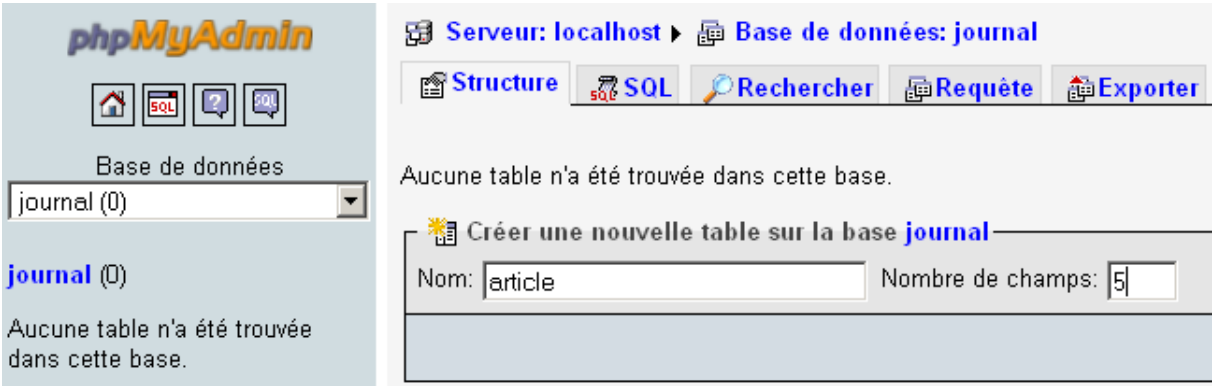
Les 2 bases existantes « **information_schema** » et « **mysql** » servent au fonctionnement interne de mysql.

Nous allons créer une base de données nommée « journal ».



CREER UNE TABLE

Une fois la base de données journal créée, il n'existe aucune table à l'intérieur, nous allons donc en créer une, nous la nommerons «article ».

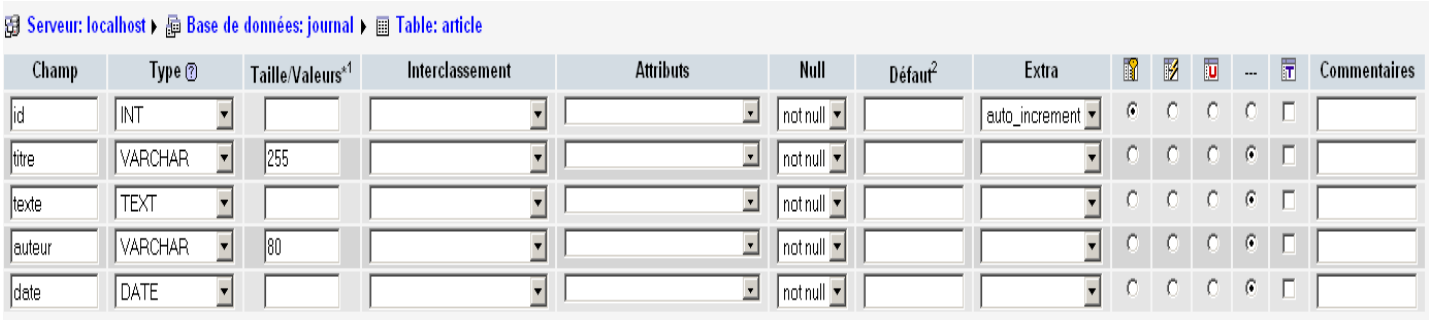


Il ne faut pas oublier de cliquer sur « exécuter », la table n'est pas immédiatement créée, il faut maintenant indiquer le nom des champs et les données qu'ils peuvent contenir.

CREER DES CHAMPS

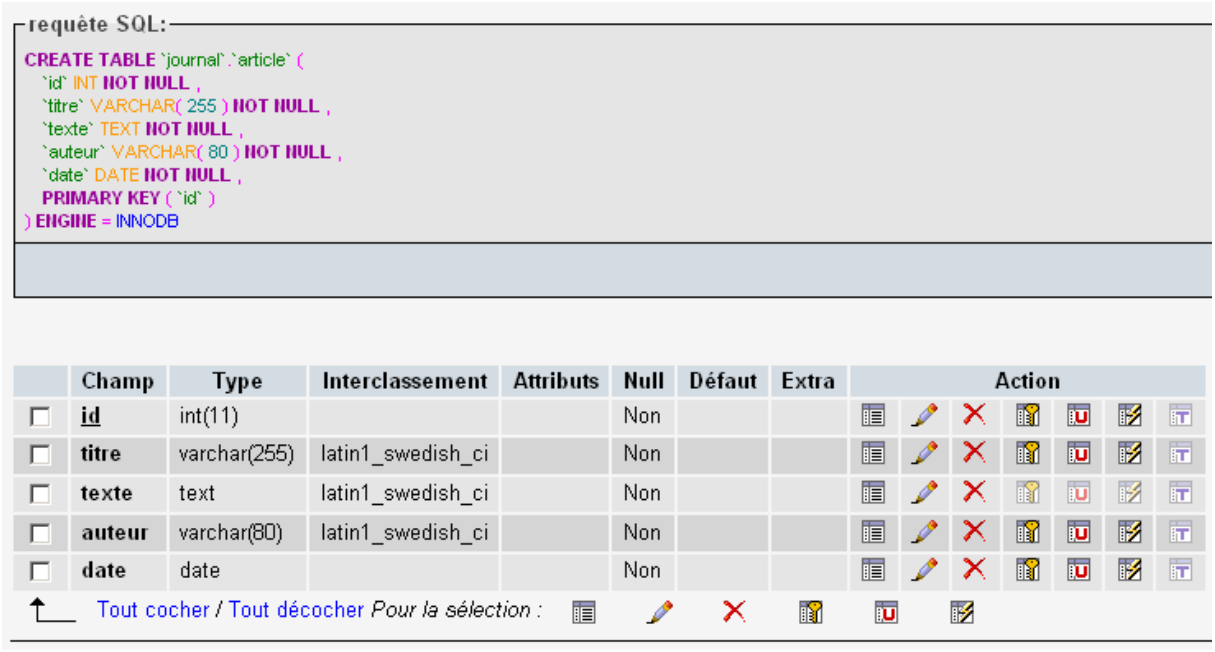
Nous allons créer les 5 champs suivants pour cette table :

- Id : INT, Auto Incrément (ce champ sert à différencier les enregistrements dans la table en cas de doublons)
- Titre : VARCHAR(255), sera le titre d'un article
- Texte : TEXT, le texte complet de l'article
- Auteur : VARCHAR(60), pour le nom de l'auteur
- Date : DATE, la date de publication par exemple.



Chaque champ représente une colonne dans la table. Nous avons demandé 5 champs, il y aura donc 5 colonnes.

ONGLET STRUCTURE



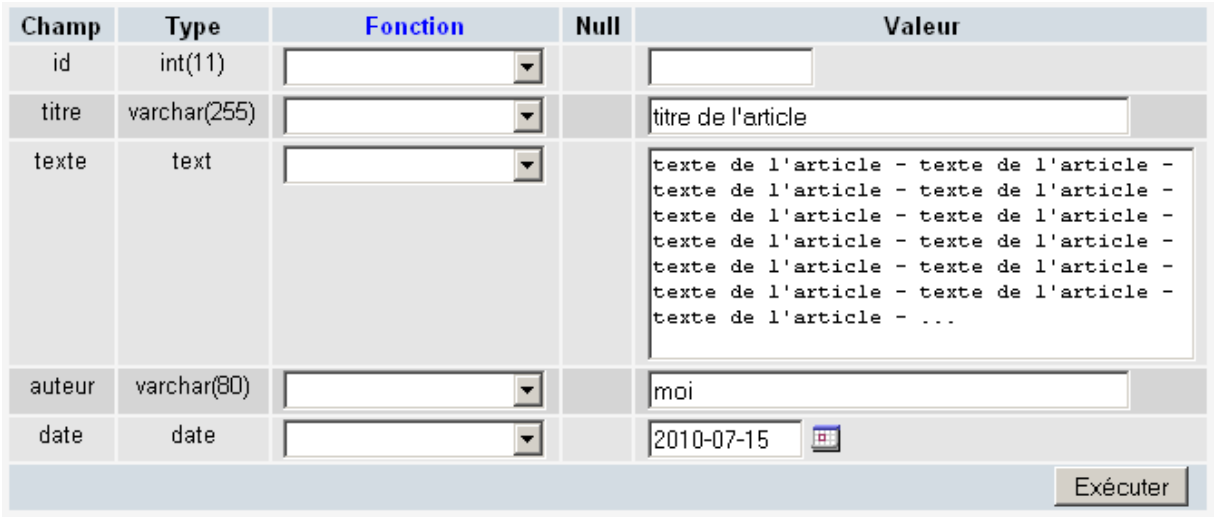
Nous pouvons apercevoir le code sql qui est généré pour la création de la table (c'est phpmyadmin qui nous le communique) ainsi que le récapitulatif de notre table.

Dans cette rubrique, nous pouvons également ajouter des champs en cas d'oubli ou de changement (cela évite d'avoir à détruire la table puis la recréer).

ONGLET SQL

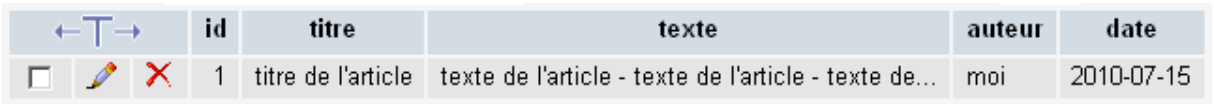
De la même manière qu'avec la console, cette rubrique nous permet d'exécuter une requête sur la table.

ONGLET INSERER



Cette rubrique permet d'enregistrer une entrée dans la table « article ».

ONGLET AFFICHER



←T→	id	titre	texte	auteur	date
☐ ✎ ✕	1	titre de l'article	texte de l'article - texte de l'article - texte de...	moi	2010-07-15

Cette rubrique nous permet de visualiser les données contenues dans notre table « article ».

ONGLET RECHERCHER

Permet d'aider l'élaboration de requête afin de récolter des informations sur des enregistrements.
Un onglet ***requête*** permet également de générer des requêtes plus approfondies.

ONGLET EXPORTER/IMPORTER

Ces rubriques permettent pour l'une d'exporter sa base de données ainsi que son contenu (dans un fichier texte ou autre) afin d'en faire une sauvegarde ou de l'importer sur son serveur de production.
Pour l'autre (l'onglet import) permet de recevoir une base de données avec déjà plusieurs informations sous certains formats de fichiers.
Afin d'éviter un temps (modifiable) de réponse trop court du serveur, il est également possible de réaliser ces actions d'import/export en ligne de commande :

Export :

```
MYSQLDUMP  -u root tic_entreprise > lenomdufichier.txt
```

Import :

```
MYSQL -u root lenomdelabase < lenomdufichier.sql
```

ONGLET OPÉRATIONS

Permet d'effectuer différentes opérations sur la base de données telle que changer le nom de la base de données courante, ordonner la table courante, copier ou déplacer une table, etc.

ONGLET PRIVILEGES

Permet de lister les personnes ayants droit sur la base de données courante.

ONGLET SUPPRIMER

Vous l'aurez compris, cette rubrique permet de supprimer une base de données ou une table (avec ses enregistrements contenus à l'intérieur).

➢ Différents types de champs

LES CHAMPS NUMERIQUES

Type des champs	Description	Taille
TINYINT	Entier très petit	1 octet
SMALLINT	Entier petit compris entre -32768 et 32767, si l'option UNSIGNED est utilisée, ce nombre sera compris entre 0 et 65535.	2 octets
MEDIUMINT	Entier moyen compris entre -8388608 et 8388607, si l'option UNSIGNED est utilisée, ce nombre sera compris entre 0 et 16777215	3 octets
INT	Entier standard compris entre –2 147 483 648 et 2147 483 647. Si l'option UNSIGNED est utilisée, ce nombre sera compris entre 0 et 4 294 967 295	4 octets
BIGINT	Entier grand	8 octets
FLOAT	Décimal de simple précision	4 octets
DOUBLE, REAL	Décimal de double précision	8 octets
DECIMAL (entier,décimal)	Réel, définissez la longueur de chacune des deux parties.	variable

Le nombre d'octets détermine la place que prend chaque champ et influe, donc, sur la grandeur des nombres que l'on peut y stocker.

LES CHAINES DE CARACTERES

Type des champs	Description	Taille
CHAR (n)	Chaîne de caractères variables. M peut être compris entre 1 et 255.	En fonction de ce qui est défini
VARCHAR(M)	Chaîne de caractères variables. M peut être compris entre 1 et 255.	255 caractères. maximum
TINYBLOB, TINYTEXT	Petite zone de texte. Objet d'une longueur maximale de 255 caractères, TINYTEXT aura un contenu de type ASCII (casse insensible) et TINYBLOB aura un contenu de type binaire (casse sensible).	255 caractères. maximum
BLOB, TEXT	Zone de texte standard Objet d'une longueur maximale de 65535 caractères, TEXT aura un contenu de type ASCII (casse insensible) et BLOB aura un contenu de type binaire (casse sensible).	65 535 caractères. maximum
MEDIUMBLOB, MEDIUMTEXT	Zone de texte moyenne. Objet d'une longueur maximale de 16777216 caractères, MEDIUMTEXT aura un contenu de type ASCII (casse insensible) et MEDIUMBLOB aura un contenu de type binaire (casse sensible).	16 millions caractères. maximum
LONGBLOB, LONGTEXT	Grande zone de texte. Objet d'une longueur maximale de 4294967295 caractères, LONGTEXT aura un contenu de type ASCII (casse insensible) et LONGBLOB aura un contenu de type binaire (casse sensible).	4 milliards caractères. maximum
ENUM('valeur','valeur2',...)	Une valeur parmi plusieurs Objet texte qui ne peut avoir qu'une des valeurs 'valeur','valeur2',...	65535 valeurs max.
SET('valeur','valeur2',...)	Une ou plusieurs valeurs parmi plusieurs. Objet texte qui peut avoir une ou plusieurs des valeurs 'valeur','valeur2',...	64 valeurs max.

Il faut noter que les champs de type CHAR et VARCHAR ne sont pas sensibles à la casse. Autrement dit, lors d'une recherche, "texte" sera identique à "TexTe" . En effet, pour rendre ces types de colonnes sensibles à la différence entre majuscule et minuscule, il faut ajouter l'argument BINARY dans la définition du champ (ex : VARCHAR (25) BINARY).

Les quatre types de champs suivants (BLOB et TEXT) n'ont, quant à eux, aucun argument. Ils sont utilisés pour stocker tous types de données (texte, images, etc.). Il faut noter qu'une colonne de type BLOB est sensible à la casse tandis qu'une colonne de type TEXT ne l'est pas.

LES CHAMPS DE TYPES DATE ET HEURE

Type des champs	Description	Taille
DATE	Date (ex: 2010-08-23)	3 octets
TIME	Heure (ex: 23:44:05)	3 octets
DATETIME	Date et heure (ex: 2000-08-24 23:44:05)	8 octets
YEAR	Année (ex: 2010)	1 octet

Le type DATE est utilisé pour manipuler simplement une date, sans l'heure. MySQL retourne et affiche les valeurs de type DATE au format 'YYYY-MM-DD'
MySql représente les dates ainsi : l'année, suivie du mois, puis du jour. Le 23 août 2010 est donc représenté sous la forme "2010-08-23 ".

Le type DATETIME est utile pour manipuler en même temps une date et une heure. MySQL retourne et affiche les valeurs de type DATETIME au format 'YYYY-MM-DD HH:MM:SS'.

Le type TIMESTAMP est utilisé automatiquement lors de requête, avec la valeur courante de date et d'heure. Il est aussi utilisé pour calculer des différences entre deux dates.

Chacun de ces différents types a un argument optionnel permettant de changer légèrement le formatage.