



LE LANGAGE SQL

STRUCTURED QUERY LANGUAGE

Concevoir et utiliser une base de données relationnelle.

Concepteur Développeur d'Applications

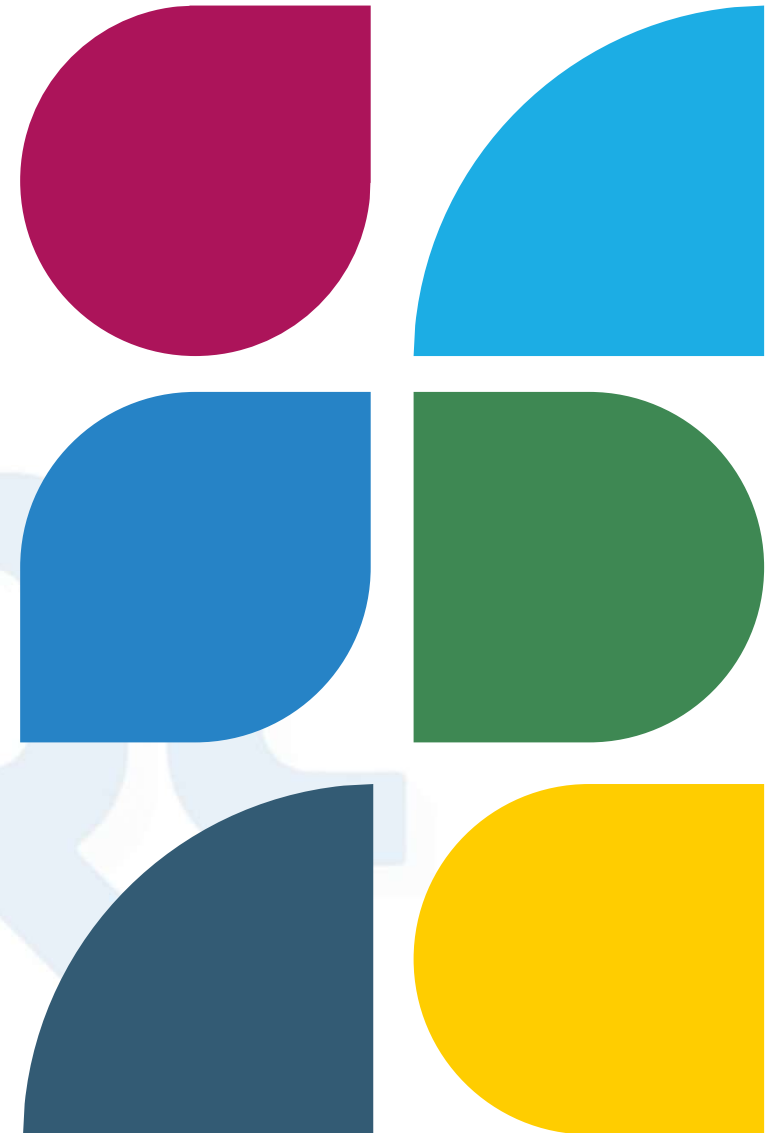


TABLE DES MATIÈRES

1. INTRODUCTION
2. LE SYSTÈME DE GESTION DE BASE DE DONNÉES
3. SQL
4. MYSQL
5. MYSQL - SYNTAXE
6. MYSQL - MANIPULATION DE SCHEMA [BASE]
7. MYSQL - SCHÉMA ET DATABASE
8. MYSQL - MANIPULATION DE TABLES
9. MYSQL - LES VUES
10. MYSQL - TABLES TEMPORAIRES
11. MYSQL - CLÉS ÉTRANGÈRES
12. MYSQL - MANIPULATION DE DONNÉES
13. MYSQL - LES REQUÊTES
14. MYSQL - GROUP BY ET HAVING
15. MYSQL - REQUÊTE AVEC SOUS-REQUÊTE
16. MYSQL - LES JOINTURES
17. MYSQL - EXEMPLE DE JOINTURES
18. MYSQL - DÉMARCHE POUR L'ÉCRITURE DE SELECT
19. MYSQL - AUTRES ÉLÉMENTS DU LANGAGE
20. MYSQL - CRÉATION D'INDEX
21. MYSQL - TRANSACTIONS
22. MYSQL - PROCÉDURES STOCKÉES
23. MYSQL - CURSEURS
24. MYSQL - LES TRIGGERS
25. MYSQL - GESTION DES ERREURS
26. MYSQL - REGEX
27. MYSQL - GESTION DES UTILISATEURS
28. MYSQL - NOTION DE SÉCURITÉ

INTRODUCTION

L'OUTIL BASE DE DONNÉES (BDD)

Historique

Face à l'augmentation d'informations que les entreprises doivent gérer et partager, il apparaît dans les années 1960, la notion de concept de base de données (BDD).

Définition

Une BDD est un ensemble structuré de données enregistré sur un support accessibles pour satisfaire simultanément plusieurs utilisateurs de façon sélective et en un temps opportun.

La structure de l'ensemble requiert une description rigoureuse appelée **SCHEMA**.



LE SYSTÈME DE GESTION DE BASE DE DONNÉES - SGBD

LE SYSTÈME DE GESTION DE BASE DE DONNÉES

Historique

Les SGBD ont près de 50 ans d'histoires.

A la fin des années 1960, apparaissent les 1^{er} SGBD (structure de type graphe et langage navigationnels).

Dans les années 1970, basée sur la théorie mathématique des relations, apparaissent les 1^{er} SGBD/R (R pour relation).

Objectifs de l'approche BDD

4 objectifs principaux :

- Intégration et corrélation
- Flexibilité
- Disponibilité
- Sécurité

12 règles de Cood

- Les 12 règles de Cood sont un ensemble de règles édictées par Edgard F. Cood conçues pour définir ce qui est exigé d'un SGBD afin qu'il puisse être considéré comme SGDB/R



LES RÈGLES DE COOD

DÉTAILS DES 4 OBJECTIFS PRINCIPAUX

Intégration et corrélation

un "réservoir" commun (intégration) est constitué, représentant une modélisation (corrélation) aussi fidèle que possible de l'organisation réelle de l'entreprise.

Toutes les applications puisent dans ce réservoir les données les concernant, évitant les duplications.

Flexibilité

Dans le cas des BDD, cette notion porte généralement le nom d'indépendance.

- **Indépendance physique** : le changement de support n'aura pas d'impact que l'accès aux données.
- **Indépendance logique** : Les changements au niveau logique (tables, colonnes, rangées, etc) ne doivent pas exiger un changement dans l'application basée sur les structures.
- **Indépendance des stratégies d'accès** : on ne doit pas prendre en charge l'écriture des procédures d'accès aux données.

DÉTAILS DES 4 OBJECTIFS

Disponibilité

Le choix d'une approche BDD ne doit pas se traduire par des temps de traitement plus longs que ceux des systèmes antérieurs.

En fait, tout utilisateur doit (ou devrait !) pouvoir ignorer l'existence d'utilisateurs concurrents.

Ici, on parle de performance.

Sécurité

La sécurité couvre les aspects :

- L'intégrité ou protection des accès invalide (erreurs ou pannes) et contre l'incohérence des données dans la BDD.
- La confidentialité ou protection contre l'accès non autorisé ou la modification illégale des données.

A decorative graphic on the left side of the slide. It features a blue circle in the upper left, a large green shape (a quarter-circle with a rectangular extension) in the lower left, and a magenta rounded rectangle in the center. A thin blue vertical line is positioned to the left of the blue circle.

LES DIFFÉRENTS LANGAGES D'UN SGBD

DIFFERENTS LANGAGES D'UN SGBD

Langage de Description de Données

Il permet de décrire précisément la structure de la base et le mode de stockage des données.

Dans une approche Base de Données, on effectue la description de toutes les données une fois pour toutes :

- elle est constituée de l'ensemble des tables et dictionnaires de la base, son schéma.

En particulier, le LDD précise :

- la structure logique des données (nom, type, contraintes spécifiques...),
- la structure physique (mode d'implantation sur les supports, mode d'accès), la définition des sous-schémas ou "vues".

Langage de Manipulation de Données

Il convient de rappeler que l'utilisation d'une BDD suppose un grand nombre d'utilisateurs (souvent non informaticiens) ayant tous des tâches et des besoins variés auxquels le LMD doit pouvoir répondre.

On peut répertorier :

- **le langage d'interrogation ou langage de requête** : syntaxe souple, accessible aux non-spécialistes, permet la formulation de demandes utilisant des critères variés et combinés. [SQL en un exemple typique.](#)
- **le langage hôte** : Pour les traitements réguliers ou d'importants volumes d'informations : une interface permettant l'utilisation de la base à l'aide des langages généraux (COBOL, C, Basic, Java...).

RÔLE DU SGBD

Rappelons succinctement les différentes fonctions assumées par un SGBD :

- description de la structure de la base ([schéma](#))
- organisation du stockage physique
- manipulation des informations (sélection, extraction, mise à jour)
- protection (sécurité)

Pour personnaliser de façon fiable les accès à la base, il convient d'identifier l'utilisateur (login et mot de passe) et de vérifier qu'il est autorisé à effectuer sur les données les traitements qu'il demande (contrôle des droits d'accès par des ACL [Access Control List]).

L'essentiel de la mise en œuvre de ces fonctions revient à une personne (ou une équipe) appelée [administrateur de la BDD](#) ou [DataBase Administrator](#).

Les LMD se répartissent en 2 catégories principales :

■ Langages navigationnels :

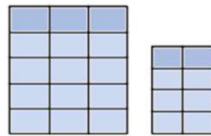
- On les rencontre avec les SGBD '**hiérarchiques**' ou '**réseaux**'. Les requêtes du langage (ou questions) décrivent les chemins d'accès aux différentes données, celles-ci étant généralement chaînées entre elles.

■ Langages algébriques (ex : SQL)

- On les rencontre avec les SGBD relationnels. Ils utilisent, pour fournir des résultats aux requêtes, les opérateurs de l'algèbre relationnelle.

LANGAGES ALGÈBRIQUES

- **le modèle relationnel** (SGBDR, *Système de gestion de bases de données relationnelles*) : les données sont enregistrées dans des tableaux à deux dimensions (lignes et colonnes). La manipulation de ces données se fait selon la théorie mathématique des relations



Ce sera ce type de modèle que nous allons étudier et utiliser.

Dans ce modèle :

- une relation est un ensemble de tuples (collections non ordonnées de valeurs connues avec des noms) ou uplets (collections ordonnées de valeurs connues avec des noms) dont l'ordre est sans importance : chaque tuple est unique.
- les colonnes de la table sont appelées attributs ou champs. Leur ordre est défini lors de la création de la table.
- une clé est un ensemble ordonné d'attributs qui caractérise un tuple. Une clé primaire le caractérise de manière unique, à l'inverse d'une clé secondaire.



STRUCTURED QUERY LANGUAGE - SQL

COMPOSITION

SQL est un langage déclaratif, il n'est donc pas à proprement parlé un langage de programmation, mais plutôt une interface standard pour accéder aux bases de données.

Il est composé de quatre sous ensemble :

- Le LDD (Langage de Définition de Données) pour créer et supprimer des objets dans la base de données (tables, contraintes d'intégrité, vues, etc..).
 - exemple : CREATE, DROP, ALTER
- Le LCD (Langage de Contrôle de Données) pour gérer les droits sur les objets de la base (création des utilisateurs et affectations de leurs droits).
 - exemple : GRANT, REVOKE
- Le LMD (Langage de Manipulation de Données) pour la recherche, l'insertion, la mise à jour et la suppression de données.
 - exemple : INSERT, UPDATE, DELETE, SELECT
- Le LCT (langage de Contrôle de Transaction) pour les gestions des transaction ou annulation de modifications de données.
 - exemple : COMMIT, ROLLBACK

MYSQL - SGBD / R

MYSQL

MySQL est un Système de Gestion de Bases de Données Relationnelles (abrégé SGBD/R)

- logiciel qui permet de gérer des bases de données, et donc de gérer de grosses quantités d'informations.
- utilise pour cela le langage SQL.
- Un SGBD/R les plus connus et les plus utilisés.
- MySQL peut donc s'utiliser seul, mais est la plupart du temps combiné à un autre langage de programmation : PHP, par exemple, pour de nombreux sites web, mais aussi Java, Python, C++, et beaucoup, beaucoup d'autres.

- MySQL peut s'utiliser en ligne de commande ou avec une interface graphique (MySql Workbench).
- Pour se connecter à MySQL en ligne de commande, on utilise :
 - `mysql -u utilisateur [-h hôte] -p`
- Pour terminer une instruction SQL, on utilise le caractère ;
- En SQL, les chaînes de caractères doivent être entourées de guillemets simples ' texte '
- Lorsque l'on se connecte à MySQL, il faut définir l'encodage utilisé, soit directement dans la connexion avec l'option `--default-character-set`, soit avec la commande : `SET NAMES 'utf8';`
- Ensuite, on peut utiliser le langage SQL pour manipuler les bases.

MYSQL

Fichiers de configuration

Si l'on veut garder la même configuration en permanence malgré les redémarrages de serveur et pour toutes les sessions, il existe une solution plus simple que de démarrer chaque fois le logiciel avec les options désirées : utiliser **les fichiers de configuration**.

Emplacement	Commentaire
WINDIR\my.ini, WINDIR\my.cnf	WINDIR est le dossier de Windows. Généralement, il s'agit du dossier C:\Windows. Pour vérifier, il suffit d'exécuter la commande suivante (dans la ligne de commande Windows) : echo %WINDIR%
C:\my.ini ou C:\my.cnf	
INSTALLDIR\my.ini ou INSTALLDIR\my.cnf	INSTALLDIR est le dossier dans lequel MySQL a été installé.

MYSQL : LES MOTEURS DU SGBD

MyISAM

Le moteur historique de MySQL, il est d'ailleurs utilisé par défaut.

- Rapide tant en lecture qu'en écriture, et il est très bien intégré à MySQL
- Ne gère pas les relations, c'est-à-dire qu'on ne peut pas définir de contrainte d'intégrité référentielle (clé étrangère / foreign key).

InnoDB

Apparu par la suite, la plus importante différence avec MyISAM

- Moteur relationnel : il permet de créer des contraintes d'intégrité, tout comme d'autres SGBD comme PostgreSQL, SQL Server ou Oracle.

À première vue, on pourrait se dire qu'il vaut mieux utiliser InnoDB que MyISAM

- Point important avant de se décider : La Performance

MyISAM et InnoDB

- sensiblement la même performance en lecture,
- en écriture, InnoDB est plus lent que MyISAM.
- InnoDB occupent plus de place de stockage que MyISAM.

TYPE DE DONNÉES DE MYSQL

- MySQL définit plusieurs types de données
 - des numériques entiers,
 - des numériques décimaux,
 - des textes alphanumériques,
 - des chaînes binaires alphanumériques
 - des données temporelles.
- Il est important de toujours utiliser le type de donnée adapté à la situation.
- SET et ENUM sont des types de données qui n'existent que chez MySQL. Il vaut donc mieux éviter de les utiliser.

```

TINYINT (1o : -127+128)
SMALLINT (2o : +-65 000)
MEDIUMINT (3o : +-16 000 000)
INT (4o : +- 2 000 000 000)
BIGINT (8o : +- 9 trillions)
  Intervalle précis :  $-(2^{(8*N-1)}) \rightarrow (2^{(8*N)})-1$ 
  /\ INT(2) = "2 chiffres affichés" -- ET NON PAS "nombre à 2 chiffres"

FLOAT(M,D) DOUBLE(M,D) FLOAT(D=0->53)
  /\ 8,3 -> 12345,678 -- PAS 12345678,123!

TIME (HH:MM)
YEAR (AAAA)
DATE (AAAA-MM-JJ)
DATETIME (AAAA-MM-JJ HH:MM; années 1000->9999)
TIMESTAMP (comme date, mais 1970->2038, compatible Unix)

VARCHAR(ligne)
TEXT (multi-lignes; taille max=65535)
BLOB (binaire; taille max=65535)

Variantes :
TINY (max=255)
MEDIUM (max=~16000)
LONG (max=4Go)
  Ex : TINYTEXT, LONGBLOB, MEDIUMTEXT

ENUM ('valeur1', 'valeur2', ...) -- (default NULL, ou '' si NOT NULL)

```

MYSQL - SYNTAXE

MOTS RÉSERVÉS ET CONVENTION DE NOMMAGES

Mots réservés

Il existe tous un ensemble de mots réservés que nous pouvons donc pas utiliser comme nom de table, nom d'attributs, variables etc...

Liste des mots réservés

Dans tous les cas, le SGBD vous l'indiquera lors de l'écriture vos requêtes. Il y a un moyen d'échappement mais si on peut s'éviter des complications...

Convention de nommages.

Comme pour le code, vous devez convenir d'une convention de nommage pour l'écriture de vos tables et de leurs attributs.

Les conventions que nous avons déjà utilisées, soit PascalCase, camelCase etc... sont à conserver.

Cependant, une convention que j'aime utiliser pour avoir une facilité de lecture de mes requêtes SQL :

PERSONNE	
PER_ID	COUNTER
PER_NOM	VARCHAR(50)
PER_PRENOM	VARCHAR(50)

LES VARIABLES UTILISATEUR

Vous avez la possibilité d'utiliser, de déclarer des variables.

Il existe trois types de variables dans le langage SQL.

La commande `SHOW VARIABLES;` permet l'affichage dans MySQL Workbench, de l'ensemble des variables et de leurs valeurs. (également disponible en ligne de commandes).

- Variables locales

- Forcément de type scalaires (entier, décimale, chaînes, booléen)
- Pour un tableau, il faut créer une table temporaire ou concaténer chaque ligne...
- Elles sont déclarées après DECLARE avec leur nom, leur type, et éventuellement la valeur par défaut.
 - `DECLARE MaVariable INT DEFAULT 1;`

- Variables de session

- Leur nom commence par `@` et dure le temps du thread.
- Déclarer par SET, ou SELECT
 - `SET @date = 'date'; SELECT @test := 2;`
- Une variable définie dans la liste de champs ne peut être utilisé comme une condition.

- Variables globales

- Visible pour tous les utilisateurs et est précédée de `@@`
- peuvent modifier les fichiers de configurations pendant la session, donc nécessaire de préciser le critère définitif ou éphémère avec `SET GLOBAL` ou `SET SESSION`

LES ALIAS

Les Alias

Une expression ou une colonne peut être baptisée avec **AS**.

Cet alias est utilisé comme nom de colonne et peut donc être nommé dans les clauses des requêtes.

Il peut servir comme raccourci à un nom de table (utile pour les jointures)

Ces Alias fonctionnent avec ORDER BY, GROUP BY, HAVING mais pas WHERE

Exemple :

```
SELECT
  p.nom AS parent,
  e.nom AS enfant,
  MIN((TO_DAYS(NOW())-TO_DAYS(e.date_naissance))/365) AS agemini
FROM
  personne AS p
LEFT JOIN
  personne AS e
ON
  p.nom=e.parent WHERE e.nom IS NOT NULL
GROUP BY
  parent HAVING agemini > 50 ORDER BY p.date_naissance;
```


LA VALEUR NULL

SQL possède une valeur pour représenter l'absence de valeur : **NULL**.

Il peut être assigné à des colonnes TEXT, INTEGER ou autres.

Attention, une colonne déclarée NOT NULL ne pourra pas en contenir.

NULL ne doit pas être entouré d'apostrophes ou de guillemets, ou bien il désignera une chaîne de caractères.

Il faudra donc dans votre futur codage prendre en compte le renvoi de la valeur NULL dans les résultats de vos requêtes.

```
INSERT into Singer
(F_Name, L_Name, Birth_place, Language)
values
("", "Homer", NULL, "Greek"),
("", "Sting", NULL, "English"),
("Jonny", "Five", NULL, "Binary");
```



MANIPULATION DE SCHEMA [BASE]

NOTRE SCHÉMA DEVIENT UNE BASE DE DONNÉES

CRÉATION, SUPPRESSION, COPIE ET BACKUP

Création et Suppression

- `CREATE DATABASE Nom_de_la_base;`
 - Permet de créer un nouveau schéma soit une nouvelle base de données.
- `DROP DATABASE Nom_de_la_base;`
 - Permet de détruire une nouvelle base de données.

Copie et Backup

- `mysqldump`
 - Peut sauvegarder les bases. Il suffit de réinjecter son résultat dans une autre base

Commande exécuter en console en étant déconnecter de la base de donnée [backup + restauration] :

```
mysqldump -u user -p pass nom_de_la_base > sauvegarde.sql
```

```
mysql nom_base < chemin_fichier_de_sauvegarde.sql
```

Depuis MySQL Workbench ou PHP MyAdmin, il suffit d'utiliser la fonction d'export où vous pouvez dès lors sauvegarder la base avec ou sans les données.

UTILISATION ET WARNINGS

Utilisation d'une base

Il faut penser avant tout de s'assurer que vous êtes sur le bon schéma :

`USE nom_de_la_base;`

Dans MySQL Workbench, un clic droit sur le schéma, puis `set as default schema` permet de sélectionner le schéma sur lequel on souhaite travailler.

Affichage des warnings

Que ce soit sur Workbench ou en ligne de commandes, on a la possibilité d'afficher les warnings :

- `SHOW WARNINGS;`

A decorative graphic on the left side of the slide. It features a blue circle in the upper left, a large green shape in the lower left, and a magenta rounded rectangle in the center. A thin blue vertical line is positioned to the left of the blue circle.

MYSQL - SCHÉMA ET DATABASE

DIFFÉRENCES

SCHEMA VS DATABASE

La différence fondamentale entre **DATABASE** et **SCHEMA** est que la base de données est manipulée régulièrement tandis que schéma n'est pas modifié fréquemment.

- **SCHEMA** est la définition structurelle de la base de données tandis que la **DATABASE** est la collection de données organisées et interdépendantes.
- **DATABASE** contient le schéma et les enregistrements des tables, mais **SCHEMA** inclut les tables, le nom de l'attribut, le type d'attribut, les contraintes, etc.
- L'instruction **DDL (Data Definition Language)** est utilisée pour générer et modifier le schéma tandis que **DML (Data Manipulation Language)** est utilisé pour la manipulation des données dans la database.
- Le schema n'utilise pas de mémoire à des fins de stockage, alors que la database le fait.

SCHEMA CONTRE DATABASE

MYSQL

Dans MySQL, **DATABASE** et **SCHEMA** peuvent être utilisés de manière interchangeable.

Vous pouvez utiliser **SCHEMA** au lieu de **DATABASE** et vice versa lors de l'écriture de requêtes SQL dans MySQL.

Voir l'exemple suivant - les deux requêtes créeront une database.

- `CREATE DATABASE database_name_one;`
- `CREATE SCHEMA database_name_two;`

Oracle - PostgreSQL

Un schéma contient un groupe de tables.

Une base de données contient un groupe de schémas.



MYSQL- MANIPULATION DE TABLES

NOS ENTITÉS DEVENUES TABLES

CRÉATION

Lorsqu'on crée une table, on doit l'identifier par son nom et définir sa structure soit les colonnes qui la composent.

La clause **PRIMARY KEY** est utilisée pour identifier la clé primaire de la table.

- obligatoires et uniques.

Cette contrainte permet de s'assurer du respect de ces caractéristiques. **À noter qu'il y a toujours une seule clé primaire par table.**

L'expression **NOT NULL** signifie qu'à l'ajout d'une nouvelle ligne dans la table, la colonne doit obligatoirement posséder une valeur. Le terme NULL dans cette colonne ne sera pas tolérée.

```
CREATE TABLE [IF NOT EXISTS] Nom_table (  
    colonne1 description_colonne1,  
    [colonne2 description_colonne2,  
    colonne3 description_colonne3,  
    ...,]  
    [PRIMARY KEY (colonne_clé_primaire)]  
)  
[ENGINE=moteur];  
// MyISAM = moteur par défaut
```

exemple :

```
CREATE TABLE Animal (  
    id SMALLINT UNSIGNED NOT NULL AUTO_INCREMENT,  
    espece VARCHAR(40) NOT NULL,  
    sexe CHAR(1),  
    date_naissance DATETIME NOT NULL,  
    nom VARCHAR(30),  
    commentaires TEXT,  
    PRIMARY KEY (id)  
)  
ENGINE=MyISAM;
```

LES CONTRAINTES

Les contraintes sont les règles appliquées aux colonnes de données d'une table.

Celles-ci sont utilisées pour limiter le type de données pouvant aller dans une table. Cela garantit l'exactitude et la fiabilité des données de la base de données.

Les contraintes peuvent être au niveau de la colonne ou de la table.

NOT NULL

- Ne peut être null

CHECK

- Vérifie la valeur saisie dans un enregistrement

DEFAULT

- Prend la valeur par défaut si non fournie par INSERT

UNIQUE

- Empêche deux enregistrements identiques dans une colonne

PRIMARY KEY

- La clé primaire

FOREIGN KEY

- La clé étrangère

MYSQL - LES VUES

LES VUES

Les vues sont des objets de la base de données, constitués d'un **nom** et d'une **requête de sélection**.

La requête SELECT stockée dans une vue :

- peut utiliser des jointures, des clauses WHERE, GROUP BY, des fonctions (scalaires ou d'agrégation), etc.
- L'utilisation de **DISTINCT** et **LIMIT** est cependant **déconseillée**.

On peut sélectionner les données à partir d'une vue de la même manière qu'on le fait à partir d'une table. On peut donc utiliser des jointures, des fonctions, des GROUP BY, des LIMIT...

Les vues permettent de simplifier les requêtes, de créer une interface entre l'application et la base de données, et/ou de restreindre finement l'accès en lecture des données aux utilisateurs.

```
-- Syntaxe
CREATE [OR REPLACE] VIEW nom_vue
AS requete_select;

-- Creation d'une vue V_Chien_Race
CREATE OR REPLACE VIEW V_Chien_race
AS SELECT id, sexe, date_naissance, nom, commentaires, espece_id, race_id,
mere_id, pere_id, disponible
FROM Chien
WHERE race_id IS NOT NULL;

-- Consultation
SELECT * FROM V_Chien_Race;
```

A decorative graphic on the left side of the slide. It features a blue circle in the upper left, a large green rounded rectangle in the lower left, and a magenta rounded rectangle in the center. A thin blue vertical line is positioned to the left of the blue circle.

MYSQL - TABLES TEMPORAIRES

TABLES TEMPORAIRES

Principe

Une table temporaire :

- n'existe que pour la session dans laquelle elle a été créée. Dès que la session se termine, les tables temporaires sont supprimées.
- créée de la même manière qu'une table normale. Il suffit d'ajouter le mot-clé **TEMPORARY** avant **TABLE**.
- créer une table (temporaire ou non) à partir de la structure d'une autre table avec **CREATE [TEMPORARY] TABLE nouvelle_table LIKE ancienne_table;**
- créer une table à partir d'une requête **SELECT** avec **CREATE [TEMPORARY] TABLE SELECT ...;**

Objectifs

1. Elles permettent de gagner en performance lorsque dans une session, on doit exécuter plusieurs requêtes sur un même set de données.
2. On peut les utiliser pour créer des données de test.
3. On peut les utiliser pour stocker un set de résultats d'une procédure stockée.

COPIE ET MODIFICATION

Copier une table

Pour obtenir la même structure (noms et types des champs, index, mais aucun enregistrement) puis dupliquer le contenu :

```
CREATE TABLE `new1` LIKE `old1`;  
  
INSERT INTO `new1` SELECT * FROM `old1`;
```

ALTER TABLE

ALTER TABLE nom_table ADD ...

- permet d'ajouter quelque chose (une colonne par exemple)

ALTER TABLE nom_table DROP ...

- permet de retirer quelque chose

ALTER TABLE nom_table CHANGE

- Peux modifier le nom, la définition (ou les deux) d'une colonne

ALTER TABLE nom_table MODIFY

- Peut **seulement** modifier la définition de la colonne.

```
ALTER TABLE Test_tuto  
|   MODIFY prenom nom VARCHAR(30) NOT NULL;  
-- Changement du type + changement du nom  
  
ALTER TABLE Test_tuto  
|   MODIFY id id BIGINT NOT NULL;  
-- Changement du type sans renommer  
  
ALTER TABLE Test_tuto  
|   MODIFY id BIGINT NOT NULL AUTO_INCREMENT;  
-- Ajout de l'auto-incrémentation  
  
ALTER TABLE Test_tuto  
|   MODIFY nom VARCHAR(30) NOT NULL DEFAULT 'BlaBla';  
-- Changement de la description (même type mais ajout d'une valeur par défaut)
```



RENOMMER ET SUPPRESSION

RENAME

Pour renommer une table, il faut préalablement retirer ses privilèges avec ALTER TABLE (DROP), puis ALTER TABLE (ADD) pour remettre les privilèges à attribuer à la nouvelle table.

Rename ne peut pas renommer les tables temporaires.

Renommage :

```
ALTER TABLE `old` RENAME `new`
```

Raccourci :

```
RENAME TABLE `old_name` TO `new_name`
```

DROP

Pour supprimer une table (enregistrements et structure), il faut "drop"

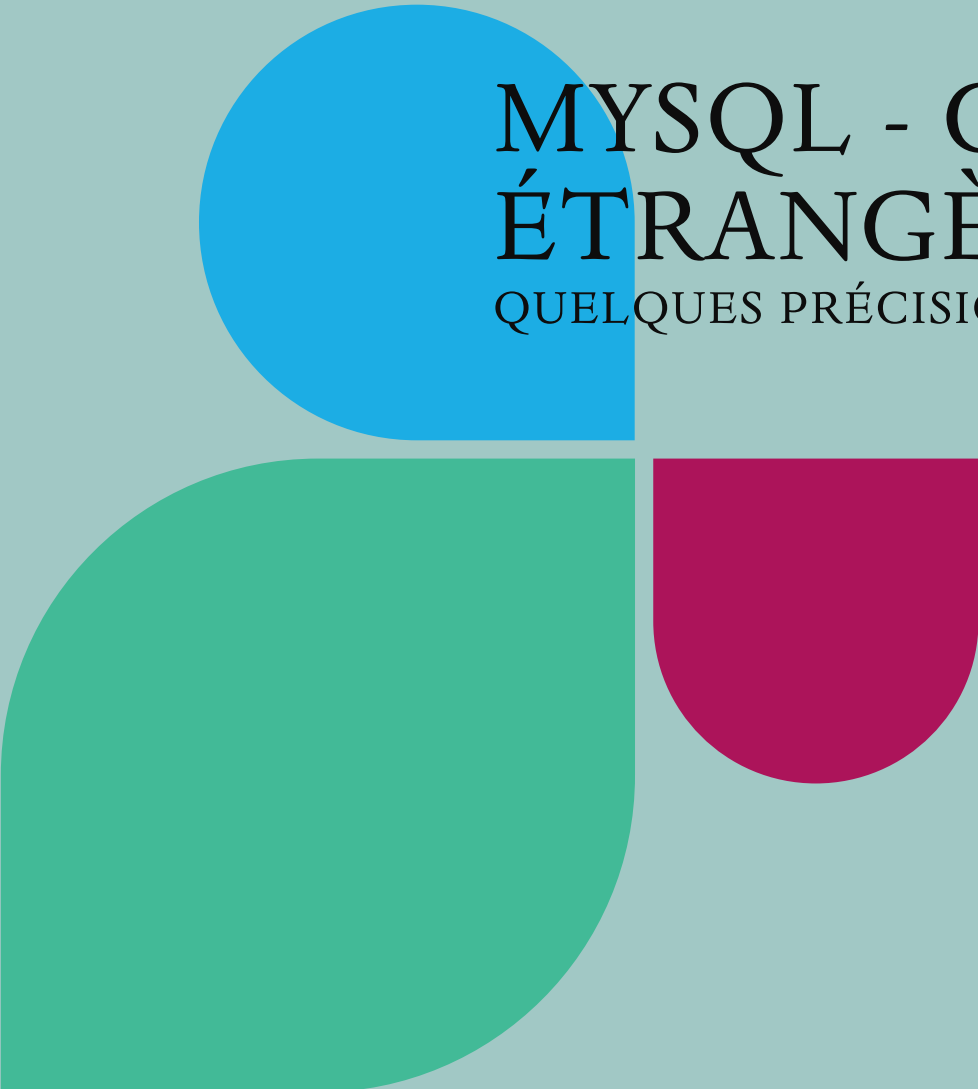
En option, on peut y ajouter une vérification d'existence.

Plusieurs :

```
DROP TABLE `table1`, `table2`, ... ;
```

Avec vérification :

```
DROP TABLE `table` IF EXISTS;
```

MYSQL - CLÉS ÉTRANGÈRES

QUELQUES PRÉCISIONS

PRINCIPE ET SYNTAXE

Les clés étrangères permettent de gérer des relations entre plusieurs tables, et garantissent la cohérence des données.

```
CREATE TABLE [IF NOT EXISTS] Nom_table (  
    colonne1 description_colonne1,  
    [colonne2 description_colonne2,  
    colonne3 description_colonne3,  
    ...,]  
    [ [CONSTRAINT [symbole_contrainte] ]  
    FOREIGN KEY (colonne(s)_clé_étrangère)  
    REFERENCES table_référence (colonne(s)_référence)]  
);  
  
CREATE TABLE Commande (  
    numero INT UNSIGNED PRIMARY KEY AUTO_INCREMENT,  
    client INT UNSIGNED NOT NULL,  
    produit VARCHAR(40),  
    quantite SMALLINT DEFAULT 1,  
    CONSTRAINT fk_client_numero          -- On donne un nom à notre clé  
    FOREIGN KEY (client)                 -- Colonne sur laquelle on crée la clé  
    REFERENCES Client(numero)           -- Colonne de référence  
);  
  
-- on peut utiliser ALTER Table pour modifier un table  
ALTER TABLE Commande  
    ADD CONSTRAINT fk_client_numero  
    FOREIGN KEY (client)  
    REFERENCES Client(numero);  
  
ALTER TABLE nom_table  
    DROP FOREIGN KEY symbole_contrainte;
```

ON DELETE ET ON UPDATE

Quand on crée une clé étrangère, il existe deux options à mettre en place afin de gérer les suppressions et les mises à jour :

- **ON DELETE** : comportement que devra avoir le SGBD qui va supprimer un enregistrement qui est référencé dans une autre table.
- **ON UPDATE** : même chose mais dans le cas d'une mise à jour qui est référencé

Ces deux options acceptent un paramètre à choisir parmi 4 ci-dessous :

- **RESTRICT** ou **NO ACTION** * : ne va rien faire. C'est à vous de faire en sorte de respecter la contrainte d'intégrité

* *Propre à MySQL donc attention si vous utilisez un autre SGBD*

- **SET NULL** : la clé étrangère reçoit la valeur NULL. Peut être utile dans le cas d'un DELETE.
- **CASCADE** : Mise à jour en cascade. *Attention : Il mettra à jour / supprimera automatiquement les enregistrements qui référencent l'enregistrement qui a été modifié / supprimé*

```
ALTER TABLE REPERTOIRE
ADD CONSTRAINT FK_REPERTOIRE FOREIGN KEY (GRO_ID)
REFERENCES GROUPE (GRO_ID) ON DELETE restrict ON UPDATE restrict;
```



MYSQL - MANIPULATION DE DONNÉES

NOS ATTRIBUTS.

INSERTION ET MODIFICATION

INSERT

```
INSERT INTO TableName (Column1, Column2, Column3)
VALUES (value1, value2, value3)

/*
Insérer un enregistrement (les valeurs sont insérées
dans l'ordre où colonnes apparaissent dans la base)
*/

INSERT INTO TableName
VALUES (value1, value2, value3)

Deux lignes :

INSERT INTO TableName
VALUES (value1, value2, value3), (value4, value5, value6)

INSERT INTO antiques
VALUES (21, 01, 'Ottoman', 200.00);

INSERT INTO antiques (buyerid, sellerid, item)
VALUES (01, 21, 'Ottoman');
```

UPDATE

```
UPDATE table
SET field1 = newvalue1, field2 = newvalue2
WHERE criteria
ORDER BY field
LIMIT n

/* mise à jour du prix de l'item chair */
UPDATE antiques SET price = 500.00 WHERE item = 'Chair';

/* mise à jour de discount */
UPDATE order SET discount=discount * 1.05
```

SUPPRESSION

DELETE

Utiliser DELETE sans clause WHERE supprime tous les enregistrements.

DELETE informe du nombre de lignes supprimées.

```
DELETE FROM nom_table
WHERE critères;

DELETE t1, t2
FROM table1 t1, table2 t2
WHERE t1.id=t2.id AND t1.value > 1;
```

TRUNCATE

En SQL, la commande TRUNCATE permet de supprimer toutes les données d'une table sans supprimer la table en elle-même.

En d'autres mots, cela permet de purger la table. Cette instruction diffère de la commande DROP qui a pour but de supprimer les données ainsi que la table qui les contient.

Note : plus rapide et moins consommateur en ressource qu'un DELETE.

Réinitialise la valeur de l'auto-incrément.

TRUNCATE n'indique pas le nombre de lignes supprimées et pas d'enregistrements dans le journal des modifications.



MYSQL - LES REQUÊTES

SÉLECTIONNER LES DONNÉES.

LES REQUÊTES - SELECT

SELECT

La syntaxe de sélection est la suivante :

```
SELECT *  
FROM nom_table  
WHERE condition  
GROUP BY champ1, champ2  
HAVING groupe condition  
ORDER BY champ  
LIMIT limite, taille;
```

Liste de champs

- Il faut spécifier les données à récupérer.
- `*` permet d'obtenir tous les champs d'une table.
- `FROM` permet de spécifier la table ciblée.
- Les calculs sont possibles.
- L'utilisation de fonctions internes à SQL sont possibles.

```
SELECT DATABASE(); -- renvoie le nom de la base courante  
SELECT CURRENT_USER(); -- l'utilisateur courant  
-- renvoie les valeurs du champ "page_id" de la table "Page".  
USE world;  
SELECT page_id FROM `Page`;  
-- idem  
SELECT `world`.`Page`.`page_id`;  
  
SELECT 1+1; -- 2  
SELECT * FROM `Page`;  
SELECT MAX(page_id) FROM `Page`; -- le nombre le plus élevé  
SELECT page_id*2 FROM `Page`; -- le double de chaque identifiant
```


LES REQUÊTES - WHERE ET ORDER BY

WHERE

Cette clause permet de filtrer les enregistrements sur une colonne.

```
SELECT colonne1, colonne2,..., colonneN
FROM nom_table
WHERE [condition]
```

Il est impossible d'utiliser le résultat d'une fonction calculée utilisée d'un SELECT dans le WHERE, car ce résultat n'est trouvé qu'à la fin de l'exécution, donc WHERE ne peut pas s'en servir au moment prévu. Pour ce faire il convient d'utiliser HAVING

ORDER BY

Il est possible de classer les résultats, par ordre croissant ou décroissant, des nombres ou des lettres.

- Par défaut l'ordre est croissant (ASC). Pour le décroissant, il faut donc le préciser DESC
- Les valeurs NULL sont considérées comme inférieures aux autres.

```
SELECT liste-colonnes
FROM nom_table
[WHERE condition]
[ORDER BY colonne1, colonne2, .. ] [ASC | DESC];
```

LES REQUÊTES - WHERE

Opérateurs logiques

Opérateur	Symbole	Signification
AND	&&	ET
OR		OU
XOR		OU exclusif
NOT	!	NON

Opérateurs de comparaisons

Opérateur	Signification
=	égal
<	inférieur
<=	inférieur ou égal
>	supérieur
>=	supérieur ou égal
<> ou !=	différent
<=>	égal (valable pour NULL aussi)

LIKE ET LES JOKERS

Si l'on cherche un mot particulier ?

L'opérateur **LIKE** est très utile, car il permet de faire des recherches en utilisant des "jokers".

Deux jokers existent pour LIKE :

% : qui représente n'importe quelle chaîne de caractères, quelle que soit sa longueur (y compris une chaîne de longueur 0) ;

_ : qui représente un seul caractère.

- Quelques exemples :
 - 'b%' cherchera toutes les chaînes de caractères commençant par 'b' ("brocoli", "bouli", "b").
 - 'b_' cherchera toutes les chaînes de caractères contenant deux lettres dont la première est 'b' ("ba", "bf", "b8").
- Comment faire si vous cherchez une chaîne de caractères contenant % ou _ ?
 - Il faut donc signaler à MySQL que vous ne désirez pas utiliser % ou _ en tant que joker, mais bien en tant que caractère de recherche.
 - Pour ça, il suffit de mettre le caractère d'échappement \, dont je vous ai déjà parlé, devant le % ou le _.

```
SELECT *  
FROM Animal  
WHERE commentaires LIKE '%\%%';
```

LES REQUÊTES - LIMIT ET DISTINCT

LIMIT :

Le nombre maximum d'enregistrements dans le résultat est facultatif, on l'indique avec le mot LIMIT.

Généralement, cela s'emploie après un ORDER BY pour avoir les maximums et minimums.

DISTINCT :

Le mot DISTINCT peut être utilisé pour supprimer les doublons des lignes du résultat.

```
/* Ce résultat retourne donc entre 0 et 10 lignes. */
SELECT * FROM `Page` ORDER BY page_id LIMIT 10;
/* Ce résultat retourne donc entre 0 et 30 lignes. */
SELECT * FROM `Page` ORDER BY rand() LIMIT 3;
/* Possible de définir une plage d'enregistrements, sachant que le premier est
le numéro zéro */
SELECT * FROM `Page` ORDER BY page_id LIMIT 10;
SELECT * FROM `Page` ORDER BY page_id LIMIT 0, 10; -- synonyme
/* On peut donc paginer les requêtes dont les résultats peuvent saturer le
serveur */
SELECT * FROM `Page` ORDER BY page_id LIMIT 0, 10; -- première page
SELECT * FROM `Page` ORDER BY page_id LIMIT 10, 10; -- seconde page
SELECT * FROM `Page` ORDER BY page_id LIMIT 20, 10; -- troisième page

/* DISTINCT */
SELECT DISTINCT * FROM `Page` -- aucun doublon
SELECT DISTINCTROW * FROM `Page` -- synonyme
SELECT ALL * FROM `Page` -- doublons (comportement par défaut)

/* Récupérer la liste de toutes les valeurs différentes d'un champ */
SELECT DISTINCT `user_real_name` FROM `Page` ORDER BY `user_real_name`

/* Sortir les différentes combinaisons de valeurs */
SELECT DISTINCT `user_real_name`, `user_editcount` FROM `Page`
ORDER BY `user_real_name`
```

IN ET NOT IN

L'opérateur logique IN dans SQL s'utilise avec la commande WHERE pour vérifier si une colonne est égale à une des valeurs comprises dans un ensemble de valeurs déterminés.

C'est une méthode simple pour vérifier si une colonne est égale à une valeur OU une autre valeur OU une autre valeur et ainsi de suite, sans avoir à utiliser de multiple fois l'opérateur OR

```
SELECT nom_colonne
FROM table
WHERE nom_colonne IN ( valeur1, valeur2, valeur3, ... )
```

Exemple

Imaginons une table "adresse" qui contient une liste d'adresse associée à des utilisateurs d'une application.

id	id_utilisateur	addr_rue	addr_code_postal	addr_ville
1	23	35 Rue Madeleine Pelletier	25250	Bournois
2	43	21 Rue du Moulin Collet	75006	Paris
3	65	28 Avenue de Cornouaille	27220	Mousseaux-Neuville
4	67	41 Rue Marcel de la Provoté	76430	Graimbouville
5	68	18 Avenue de Navarre	75009	Paris

Si l'ont souhaite obtenir les enregistrements des adresses de Paris et de Graimbouville, il est possible d'utiliser la requête suivante:

```
SELECT *
FROM adresse
WHERE addr_ville IN ( 'Paris', 'Graimbouville' )
```

Résultats :

id	id_utilisateur	addr_rue	addr_code_postal	addr_ville
2	43	21 Rue du Moulin Collet	75006	Paris
4	67	41 Rue Marcel de la Provoté	76430	Graimbouville
5	68	18 Avenue de Navarre	75009	Paris

BETWEEN

L'opérateur BETWEEN est utilisé dans une requête SQL pour sélectionner un intervalle de données dans une requête utilisant WHERE.

L'intervalle peut être constitué de chaînes de caractères, de nombres ou de dates.

L'exemple le plus concret consiste par exemple à récupérer uniquement les enregistrements entre 2 dates définies.

```
SELECT *  
FROM table  
WHERE nom_colonne BETWEEN 'valeur1' AND 'valeur2'
```

Exemple : filtrer entre 2 dates

Imaginons une table "utilisateur" qui contient les membres d'une application en ligne.

id	nom	date_inscription
1	Maurice	2012-03-02
2	Simon	2012-03-05
3	Chloé	2012-04-14
4	Marie	2012-04-15
5	Clémentine	2012-04-26

Si l'on souhaite obtenir les membres qui se sont inscrits entre le 1 avril 2012 et le 20 avril 2012 il est possible d'effectuer la requête suivante:

```
SELECT *  
FROM utilisateur  
WHERE date_inscription BETWEEN '2012-04-01' AND '2012-04-20'
```

Résultat :

id	nom	date_inscription
3	Chloé	2012-04-14
4	Marie	2012-04-15

UNION

La clause/opérateur SQL UNION est utilisée pour combiner les résultats de deux ou plusieurs instructions SELECT sans renvoyer les lignes en double.

Pour utiliser cette clause UNION, chaque instruction SELECT doit avoir :

- Le même nombre de colonnes sélectionnées
- Le même type de données et
- Les avoir dans le même ordre

Mais ils n'ont pas besoin d'être dans la même longueur.

La clause UNION produit des valeurs distinctes dans le jeu de résultats. Pour extraire les valeurs en double, UNION ALL doit être utilisée à la place de UNION.

Syntaxe - UNION

```
1 SELECT colonne1 [, colonne2 ]
2 FROM table1 [, table2 ]
3 [WHERE condition]
4
5 UNION
6
7 SELECT colonne1 [, colonne2 ]
8 FROM table1 [, table2 ]
9 [WHERE condition]
```

Syntaxe - UNION ALL

```
1 SELECT colonne1 [, colonne2 ]
2 FROM table1 [, table2 ]
3 [WHERE condition]
4
5 UNION ALL
6
7 SELECT colonne1 [, colonne2 ]
8 FROM table1 [, table2 ]
9 [WHERE condition]
```

prenom	nom	ville	date_naissance	total_achat
Léon	Dupuis	Paris	1983-03-06	135
Marie	Bernard	Paris	1993-07-03	75
Sophie	Dupond	Marseille	1986-02-22	27
Marcel	Martin	Paris	1976-11-24	39

prenom	nom	ville	date_naissance	total_achat
Marion	Leroy	Lyon	1982-10-27	285
Paul	Moreau	Lyon	1976-04-19	133
Marie	Bernard	Paris	1993-07-03	75
Marcel	Martin	Paris	1976-11-24	39

UNION

prenom	nom	ville	date_naissance	total_achat
Léon	Dupuis	Paris	1983-03-06	135
Marie	Bernard	Paris	1993-07-03	75
Sophie	Dupond	Marseille	1986-02-22	27
Marcel	Martin	Paris	1976-11-24	39
Marie	Leroy	Lyon	1982-10-27	285
Paul	Moreay	Lyon	1976-04-19	133



MYSQL - GROUP BY ET HAVING

EXPLICATIONS ET DÉMONSTRATION

ORGANISER DES DONNÉES IDENTIQUES EN GROUPES - GROUP BY ET HAVING

La clause GROUP BY en SQL permet d'organiser des données identiques en groupes à l'aide de certaines fonctions.

C'est-à-dire si une colonne particulière a les mêmes valeurs dans différentes lignes, elle organisera ces lignes dans un groupe.

- La clause GROUP BY est utilisée avec l'instruction SELECT.
- Dans la requête, la clause GROUP BY est placée après la clause WHERE.
- Dans la requête, la clause GROUP BY est placée avant la clause ORDER BY si elle est utilisée.

- Vous pouvez également utiliser certaines fonctions d'agrégation telles que COUNT, SUM, MIN, MAX, AVG, etc. sur la colonne groupée.

```
SELECT colonne1, colonne2, ... colonneN,  
fonction_agregation (nom_colonne)  
FROM tables  
[WHERE conditions]  
GROUP BY colonne1, colonne2, ... colonneN;
```

- *colonne1, colonne2, ... colonneN - spécifie les colonnes(ou expressions) qui ne sont pas encapsulées dans une fonction d'agrégation et doivent être incluses dans la clause GROUP BY.*

SELECT AGE FROM
EMPLOYES GROUP BY
AGE;

Cette requête nous renvoie 3 résultats soit 3 groupes :

- Groupe 1 - Age = 25
- Groupe 2 - Age = 29
- Groupe 3 - Age = 30

Age
25
30
29
30
30
29



Age
25
29
30

Age
25

Age
29
29

Age
30
30
30

FONCTION D'AGRÉGATIONS

Vous pouvez compter désormais le nombre d'employés de chaque groupe à l'aide de count :

```
SELECT Age, count(*) AS "Nombre d'employés"  
FROM Employes  
GROUP BY Age;
```

Age	Nombre d'employés
25	1
29	2
30	3

HAVING

Nous savons que la clause WHERE est utilisée pour imposer des conditions aux colonnes, mais que se passe-t-il si nous voulons imposer des conditions aux groupes ?

C'est ici que la clause HAVING entre en vigueur. Nous pouvons utiliser la clause HAVING pour poser des conditions afin de décider quel groupe fera partie de l'ensemble des résultats finaux.

De plus, nous ne pouvons pas utiliser les fonctions d'agrégation telles que SUM(), COUNT(), etc. avec la clause WHERE.

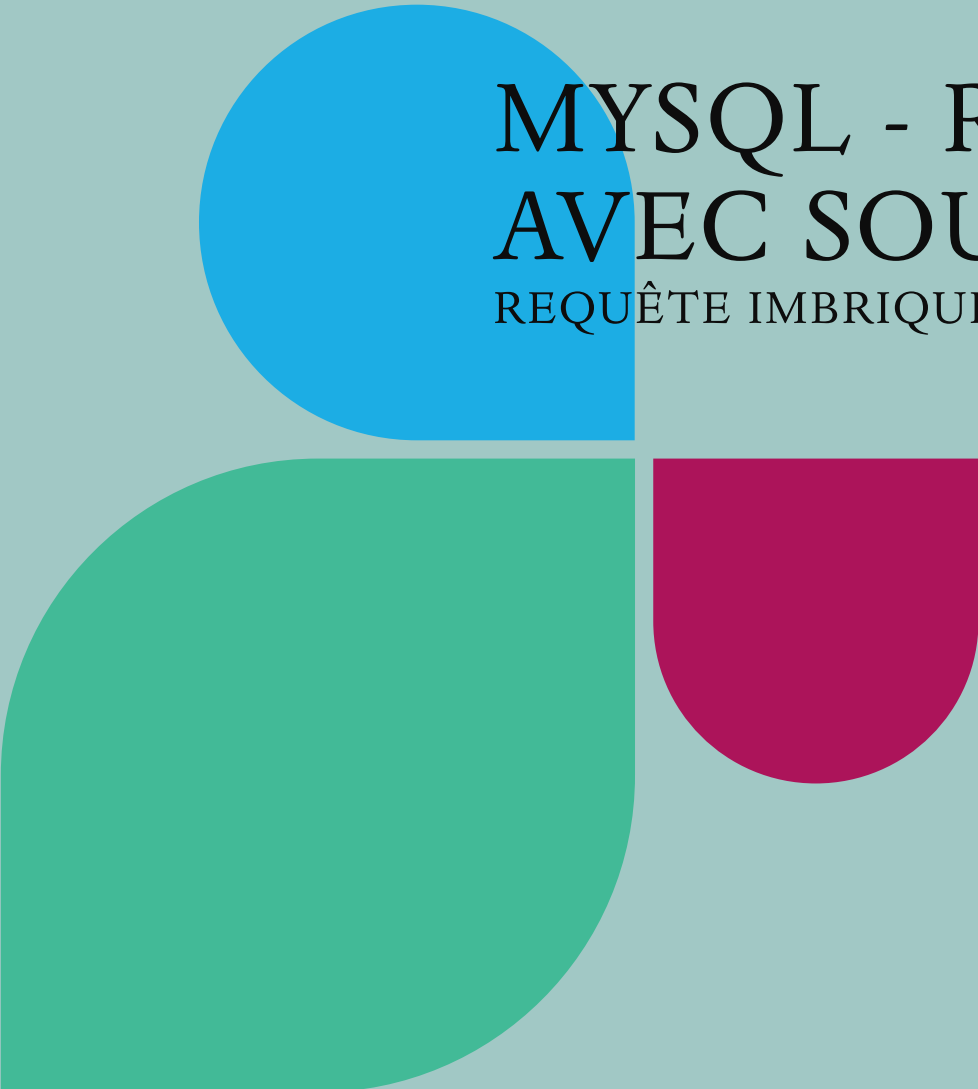
Nous devons donc utiliser la clause HAVING si nous voulons utiliser l'une de ces fonctions dans les conditions.

La requête suivante récupère les noms de département et le salaire moyen de chaque département :

```
SELECT D.Nom_dep, AVG(E.Salaire) AS "Salaire moyen"  
FROM Employes AS E INNER JOIN Departement AS D  
ON E.Dep=D.Id_dep  
GROUP BY D.Nom_dep;
```

Supposons maintenant que nous ne voulions montrer que les départements dont le salaire moyen est supérieur à 6000 ?

```
SELECT D.Nom_dep, AVG(E.Salaire) AS "Salaire moyen"  
FROM Employes AS E INNER JOIN Departement AS D  
ON E.Dep=D.Id_dep  
GROUP BY D.Nom_dep  
HAVING AVG(E.Salaire) > 6000;
```



MYSQL - REQUÊTE AVEC SOUS-REQUÊTE

REQUÊTE IMBRIQUÉE

LES SOUS-REQUÊTES EN SQL

Une sous-requête, également appelée requête imbriquée ou sous-sélection, est une requête SELECT intégrée à la clause WHERE ou HAVING d'une autre requête SQL.

Les données renvoyées par la sous-requête sont utilisées par l'instruction externe de la même manière qu'une valeur littérale serait utilisée.

Les sous-requêtes constituent un moyen simple et efficace de gérer les requêtes qui dépendent des résultats d'une autre requête.

Sous-requêtes vs jointures :

Comparées aux jointures, les sous-requêtes sont simples à utiliser et à lire. Ils ne sont pas aussi compliqués que les jointures.

Mais les sous-requêtes posent des problèmes de performances.

L'utilisation d'une jointure au lieu d'une sous-requête peut parfois vous donner un gain de performances jusqu'à 500 fois.

Si vous avez le choix, il est recommandé d'utiliser une jointure plutôt qu'une sous-requête.

```
SELECT nom_colonne [, nom_colonne ]  
FROM   table1 [, table2 ]  
WHERE  nom_colonne OPERATOR  
       (SELECT nom_colonne [, nom_colonne ]  
        FROM table1 [, table2 ]  
        [WHERE])
```

EXIST ET NOT EXISTS

Les conditions EXISTS et NOT EXISTS s'utilisent de la manière suivante :

```
SELECT * FROM nom_table  
WHERE [NOT] EXISTS (sous-requête)
```

Une condition avec EXISTS sera vraie (et donc la requête renverra quelque chose) si la sous-requête correspondante renvoie au moins une ligne.

Une condition avec NOT EXISTS sera vraie si la sous-requête correspondante ne renvoie aucune ligne.

Table commande :

c_id	c_date_achat	c_produit_id	c_quantite_produit
1	2014-01-08	2	1
2	2014-01-24	3	2
3	2014-02-14	8	1
4	2014-03-23	10	1

Table produit :

p_id	p_nom	p_date_ajout	p_prix
2	Ordinateur	2013-11-17	799.9
3	Clavier	2013-11-27	49.9
4	Souris	2013-12-04	15
5	Ecran	2013-12-15	250

RÉSULTAT

```
SELECT *  
FROM commande  
WHERE EXISTS (  
    SELECT *  
    FROM produit  
    WHERE c_produit_id = p_id  
)
```



c_id	c_date_achat	c_produit_id	c_quantite_produit
1	2014-01-08	2	1
2	2014-01-24	3	2

ALL

Dans le langage SQL, la commande ALL permet de comparer une valeur dans l'ensemble de valeurs d'une sous-requête.

En d'autres mots, cette commande permet de s'assurer qu'une condition est "égale", "différente", "supérieure", "inférieure", "supérieure ou égale" ou "inférieure ou égale" pour tous les résultats retournés par une sous-requête.

Imaginons une requête :

```
SELECT colonne1
FROM table1
WHERE colonne1 > ALL (
    SELECT colonne1
    FROM table2
)
```

Avec cette requête, si nous supposons que dans table1 il y a un résultat avec la valeur 10, voici les différents résultats de la condition selon le contenu de table2 :

- La condition est vraie si table2 contient {-5,0,+5} car toutes les valeurs sont inférieures à 10
- La condition est fausse si table2 contient {12,6,NULL,-100} car au moins une valeur est inférieure à 10
- La condition est non connue si table2 est vide



MYSQL - LES JOINTURES

PARCOURIR UNE BASE DE DONNÉES À LA
RECHERCHE DE RÉSULTATS.

PRINCIPE

On souhaite obtenir le nom latin de l'animal nommé Cartouche. Comment faire ?

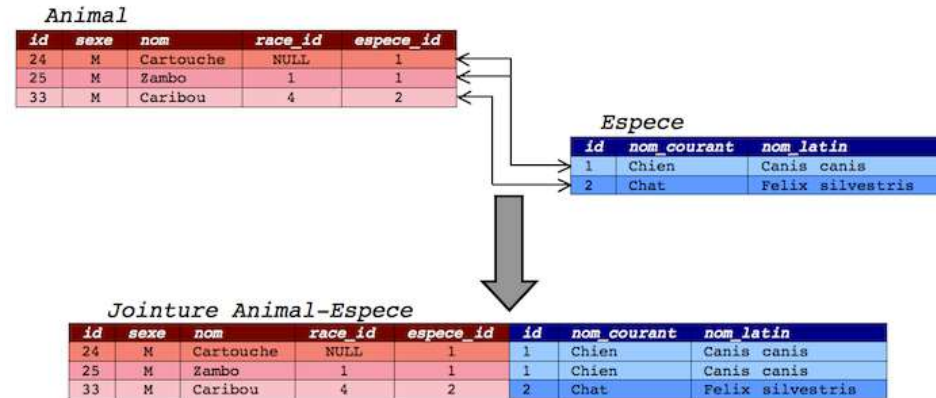
Un select sur la table Animal avec comme clause le nom "Cartouche" nous donne en résultat la ligne de l'id 24.

Un select sur la table Espece contient le nom latin..

Le point commun entre nos deux tables est :

- Espece_id de la table Animal
- Id de la table Espece

Ce qui nous donnerai alors une nouvelle table



Principe des jointures

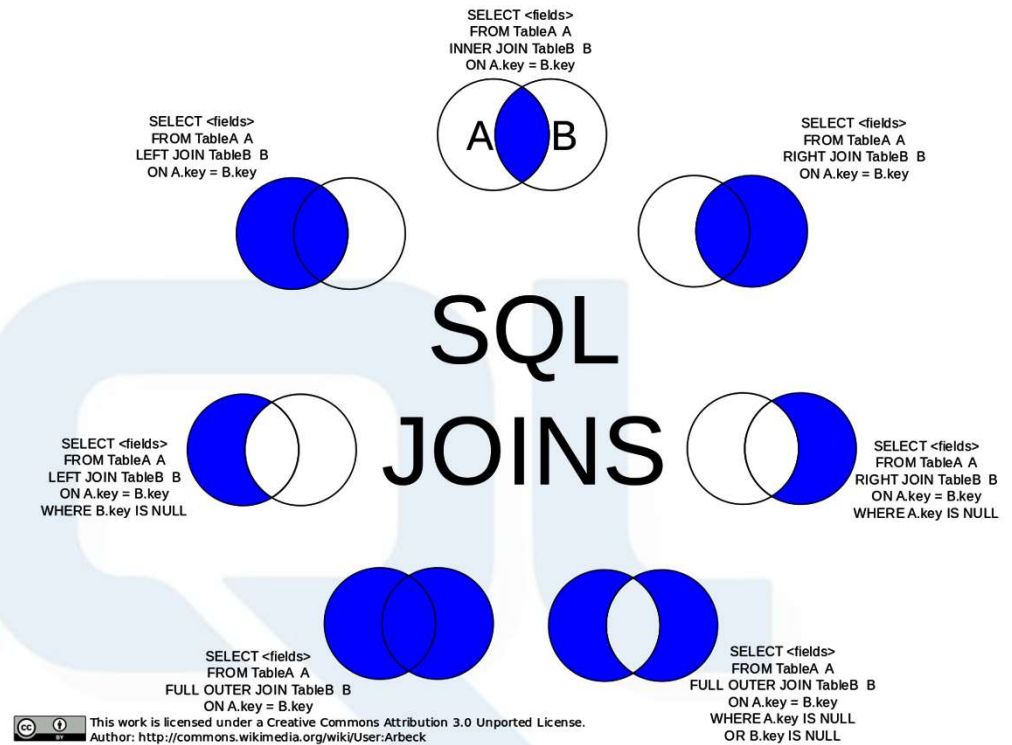
On va donc utiliser une jointure entre nos deux tables :

```
SELECT Espece.nom_latin
FROM Espece
INNER JOIN Animal
ON Espece.id = Animal.espece_id
WHERE Animal.nom = 'Cartouche';
```

DIFFÉRENTES JOINTURES

Il existe plusieurs types de jointures en SQL

La jointure par défaut est la INNER JOIN (ou JOIN)

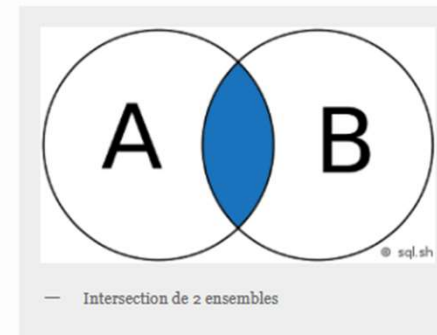


INTERNE

La jointure interne indique que l'on exige qu'il y ait des données de part et d'autre de la jointure. Donc si on fait une jointure sur la colonne a de la table A et la colonne b de la table B :

Cela retournera uniquement les lignes pour lesquelles A.a et B.b correspondent.

INNER JOIN



```
SELECT *  
FROM A  
INNER JOIN B ON A.key = B.key
```

EXTERNE

Une jointure externe permet de sélectionner également les lignes pour lesquelles il n'y a pas de correspondance dans une des tables jointes.

Deux types de jointures : par la gauche ou par la droite.

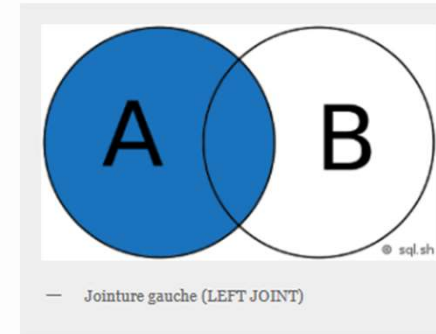
Jointures par la gauche LEFT JOIN ou LEFT OUTER JOIN

On veut toutes les lignes de la table de gauche (sauf restrictions dans une clause WHERE), même si certaines n'ont pas de correspondance avec une ligne de la table de droite.

Jointures par la droite RIGHT JOIN ou RIGHT OUTER JOIN

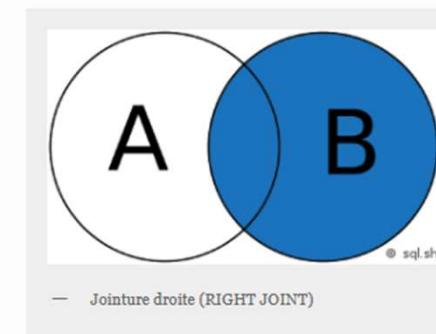
On veut toutes les lignes de la table de droite (sauf restrictions dans une clause WHERE), même si certaines n'ont pas de correspondance avec une ligne de la table de gauche.

LEFT JOIN



```
SELECT *
FROM A
LEFT JOIN B ON A.key = B.key
```

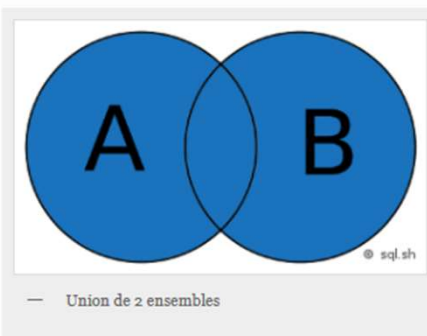
RIGHT JOIN



```
SELECT *
FROM A
RIGHT JOIN B ON A.key = B.key
```

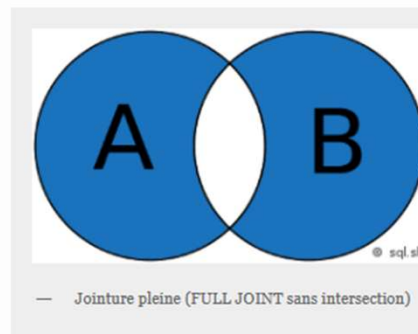
FULL

FULL JOIN



```
SELECT *
FROM A
FULL JOIN B ON A.key = B.key
```

FULL JOIN (sans intersection)



```
SELECT *
FROM A
FULL JOIN B ON A.key = B.key
WHERE A.key IS NULL
OR B.key IS NULL
```




MYSQL - EXEMPLE DE JOINTURES

COMPRENDRE PAR L'EXEMPLE

ENONCÉ

Soit la table Utilisateur

id	prenom	nom	email	ville
1	Aimée	Marechal	aime.marechal@example.com	Paris
2	Esmée	Lefort	esmee.lefort@example.com	Lyon
3	Marine	Prevost	m.prevost@example.com	Lille
4	Luc	Rolland	lucrolland@example.com	Marseille

Soit la table Commande

utilisateur_id	date_achat	num_facture	prix_total
1	2013-01-23	A00103	203.14
1	2013-02-14	A00104	124.00
2	2013-02-17	A00105	149.45
2	2013-02-21	A00106	235.35
5	2013-03-02	A00107	47.58

INNER JOIN

Pour afficher toutes les commandes associées aux utilisateurs, il est possible d'utiliser la requête suivante :

```
-- inner join
SELECT id, prenom, nom, date_achat, num_facture, prix_total
FROM utilisateur
INNER JOIN commande ON utilisateur.id = commande.utilisateur_id;
```

id	prenom	nom	date_achat	num_facture	prix_total
1	Aimée	Marechal	2013-01-23	A00103	203.14
1	Aimée	Marechal	2013-02-14	A00104	124.00
2	Esmée	Lefort	2013-02-17	A00105	149.45
2	Esmée	Lefort	2013-02-21	A00106	235.35

Le résultat de la requête montre la jointure entre les 2 tables. Les utilisateurs 3 et 4 ne sont pas affichés puisqu'il n'y a pas de commandes associés à ces utilisateurs.

L'utilisateur 5 n'étant pas existant, n'est donc pas affiché puisque la condition n'est pas vrai entre les 2 tables.

LEFT JOIN

Pour lister tous les utilisateurs avec leurs commandes et afficher également les utilisateurs qui n'ont pas effectués d'achats, il est possible d'utiliser la requête suivante :

```
-- left join
SELECT *
FROM utilisateur
LEFT JOIN commande ON utilisateur.id = commande.utilisateur_id;
```

Si on veut simplement filtrer que les utilisateurs ayant effectués des achats :

```
-- left join avec filtrage
SELECT id, prenom, nom, utilisateur_id
FROM utilisateur
LEFT JOIN commande ON utilisateur.id = commande.utilisateur_id
WHERE utilisateur_id IS NULL;
```

id	prenom	nom	date_achat	num_facture	prix_total
1	Aimée	Marechal	2013-01-23	A00103	203.14
1	Aimée	Marechal	2013-02-14	A00104	124.00
2	Esmée	Lefort	2013-02-17	A00105	149.45
2	Esmée	Lefort	2013-02-21	A00106	235.35
3	Marine	Prevost	NULL	NULL	NULL
4	Luc	Rolland	NULL	NULL	NULL

Les dernières lignes montrent des utilisateurs qui n'ont effectuée aucune commandes. La ligne retourne la valeur NULL pour les colonnes concernant les achats qu'ils n'ont pas effectués.

RIGHT JOIN

Pour afficher toutes les commandes avec le nom de l'utilisateur correspondant il est normalement d'habitude d'utiliser INNER JOIN en SQL. Malheureusement, si l'utilisateur a été supprimé de la table, alors ça ne retourne pas l'achat. L'utilisation de RIGHT JOIN permet de retourner tous les achats et d'afficher le nom de l'utilisateur s'il existe.

Pour cela il convient d'utiliser cette requête:

```
-- right join
SELECT id, prenom, nom, utilisateur_id, date_achat, num_facture
FROM utilisateur
RIGHT JOIN commande ON utilisateur.id = commande.utilisateur_id;
```

id	prenom	nom	utilisateur_id	date_achat	num_facture
1	Aimée	Marechal	1	2013-01-23	A00103
1	Aimée	Marechal	1	2013-02-14	A00104
2	Esmée	Lefort	2	2013-02-17	A00105
3	Marine	Prevost	3	2013-02-21	A00106
NUL L	NULL	NULL	5	2013-03-02	A00107

Ce résultat montre que la facture A00107 est liée à l'utilisateur numéro 5.

Or, cet utilisateur n'existe pas ou n'existe plus.

Grâce à RIGHT JOIN, l'achat est tout de même affiché mais les informations liées à l'utilisateur sont remplacé par NULL.

FULL JOIN

Il est possible d'utiliser FULL JOIN pour lister tous les utilisateurs ayant effectué ou non une vente, et de lister toutes les ventes qui sont associées ou non à un utilisateur.

La requête SQL est la suivante :

```
-- full join
SELECT id, prenom, nom, utilisateur_id, date_achat, num_facture
FROM utilisateur
FULL JOIN commande ON utilisateur.id = commande.utilisateur_id;
```

id	prenom	nom	utilisateur_id	date_achat	num_facture
1	Aimée	Marechal	1	2013-01-23	A00103
1	Aimée	Marechal	1	2013-02-14	A00104
2	Esmée	Lefort	2	2013-02-17	A00105
3	Marine	Prevost	3	2013-02-21	A00106
4	Luc	Rolland	NULL	NULL	NULL
NULL	NULL	NULL	5	2013-03-02	A00107

Ce résultat affiche bien l'utilisateur numéro 4 qui n'a effectué aucun achat.

Le résultat retourne également la facture **A00107** qui est associée à un utilisateur qui n'existe pas (ou qui n'existe plus).

Dans les cas où il n'y a pas de correspondance avec l'autre table, les valeurs des colonnes valent NULL.



MYSQL - DÉMARCHE POUR L'ÉCRITURE DE SELECT

PROPOSITION DE DÉMARCHE

QUELQUES CONSEILS

1. Décider quels sont les attributs à visualiser, les inclure dans la clause **SELECT**.
2. Les expressions présentes dans la liste de sélection d'une requête (clause **SELECT**) avec la clause **GROUP BY** doivent être des fonctions d'agrégation ou apparaître dans la liste **GROUP BY**.
3. Déterminer les tables à mettre en jeu, les inclure dans la clause **FROM**.
4. Déterminer les conditions de jointure quand plusieurs tables sont en jeu.
5. Déterminer les conditions limitant la recherche : les conditions portant sur les groupes doivent figurer dans une clause **HAVING**, celles portant sur des valeurs individuelles dans une clause **WHERE**.
6. Préciser l'ordre d'apparition des lignes de résultat dans une clause **ORDER BY**.

On peut formaliser la démarche en remplissant le tableau suivant pour aider à construire chaque requête :

Questions ?	Réponse par la requête
Quelles sont les colonnes à afficher en résultat (SELECT)	SELECT ...
De quelles tables sont issues ces colonnes (FROM)	FROM ...
Y'a-t-il des jointures (JOIN)	...
N'y-a-t-il que certaines lignes à prendre en compte (WHERE)	WHERE ...
Veut-on un résultat par "paquets" de lignes (GROUP BY)	GROUP BY ...
Veut-on voir apparaître les résultats selon un ordre précis ? (ORDER BY)	ORDER BY ...
N'y-a-t-il que certaines lignes résultats à prendre en compte ? (HAVING)	HAVING ...



MYSQL - AUTRES ÉLÉMENTS DU LANGAGE

PARCOURS DE CE QUI EST DISPONIBLE

ELÉMENTS DU LANGAGE

SQL offre divers éléments de langage permettant de créer des blocs d'instructions et des instructions conditionnelles :

commentaires

- -- commentaire
- /* ... */

bloc

- BEGIN...
END
- RETURN

LES ÉLÉMENTS DU LANGAGE

Assignment	Comparison	Logiques	Arithmétiques	Texte	Conditions
<ul style="list-style-type: none"> • SET • := • INTO 	<ul style="list-style-type: none"> • = • <> != • IS NULL • IS TRUE, IS NOT TRUE • >, <, <=, >= • BETWEEN • IN 	<ul style="list-style-type: none"> • NOT • AND • OR • XOR 	<ul style="list-style-type: none"> • +, -, /, DIV • Conversion : 1 + 0.0 donnera 1.0 	<ul style="list-style-type: none"> • LIKE 	<ul style="list-style-type: none"> • IF • CASE • WHILE condition • CASE..WHEN ..THEN..END • LOOP • REPEAT

SELECT .. INTO

Le SELECT ... INTO permet de stocker un résultat de requête dans des variables

Le positionnement de INTO est préféré en fin de la requête soit après le FROM depuis la version 8.0.20 de MySQL

Documentation officielle :

<https://dev.mysql.com/doc/refman/8.0/en/select-into.html>

Exemple :

```
-- compter le nombre d'emprunts en cours de l'adhérent
-- select .. from t into @var;
SELECT COUNT(*)
FROM pret
WHERE numab = pNumab AND dateretour IS NULL
INTO nb_emprunts ;
```



MYSQL - CRÉATION D'INDEX

INDEXER LES DONNÉES

DÉFINITION

Un index est une structure qui reprend la liste ordonnée des valeurs auxquelles il se rapporte.

Les index sont utilisés pour :

- accélérer les requêtes (notamment les requêtes impliquant plusieurs tables, ou les requêtes de recherche).
- permettre de garantir l'intégrité des données de la base. [indispensables à la création de clés, étrangères et primaires]
- accélérer les requêtes qui utilisent des colonnes indexées comme critères de recherche.

Convention de nommage

Il n'existe pas de convention de nommage spécifique sur le nom des index, juste des suggestions de quelques développeurs et administrateurs de bases de données.

Voici une liste de suggestions de préfixes à utiliser pour nommer un index :

Préfixe "PK_" pour Primary Key (traduction : clé primaire)

Préfixe "FK_" pour Foreign Key (traduction : clé étrangère)

Préfixe "UK_" pour Unique Key (traduction : clé unique)

Préfixe "UX_" pour Unique Index (traduction : index unique)

Préfixe "IX_" pour chaque autre Index

FONCTIONNEMENT

Les index ne sont pas des objets qui font partie de la théorie relationnelle. Ils sont des objets physiques qui permettent d'accélérer l'accès aux données.

Et comme ils ne sont que des moyens d'optimisation des accès, les index ne font pas non plus partie de la norme SQL.

C'est d'ailleurs pour cette raison que la syntaxe de création d'index est si différente d'une base de données à une autre.

L'index permet de pointer l'endroit de la table où se trouve une donnée, pour y accéder directement. Parfois c'est toute une plage de l'index, voire sa totalité, qui sera lue, ce qui est généralement plus rapide que lire toute la table.



Id	Id	Espèce	Sexe	Date de naissance	Nom	Commentaires
1	2	chat	NULL	2010-03-24 02:23:00	Roucky	NULL
2	1	chien	male	2010-04-05 13:43:00	Rox	Mordille beaucoup
3	3	chat	femelle	2010-09-13 15:02:00	Schtroumpfette	NULL
4	6	tortue	femelle	2009-06-13 08:17:00	Bobosse	Carapace bizarre
5	9	tortue	NULL	2010-08-23 05:18:00	NULL	NULL
6	4	tortue	femelle	2009-08-03 05:12:00	NULL	NULL
7	7	chien	femelle	2008-12-06 05:18:00	Caroline	NULL
8	8	chat	male	2008-09-11 15:38:00	Bagherra	NULL
9	5	chat	NULL	2010-10-03 16:44:00	Choupi	Né sans oreille gauche

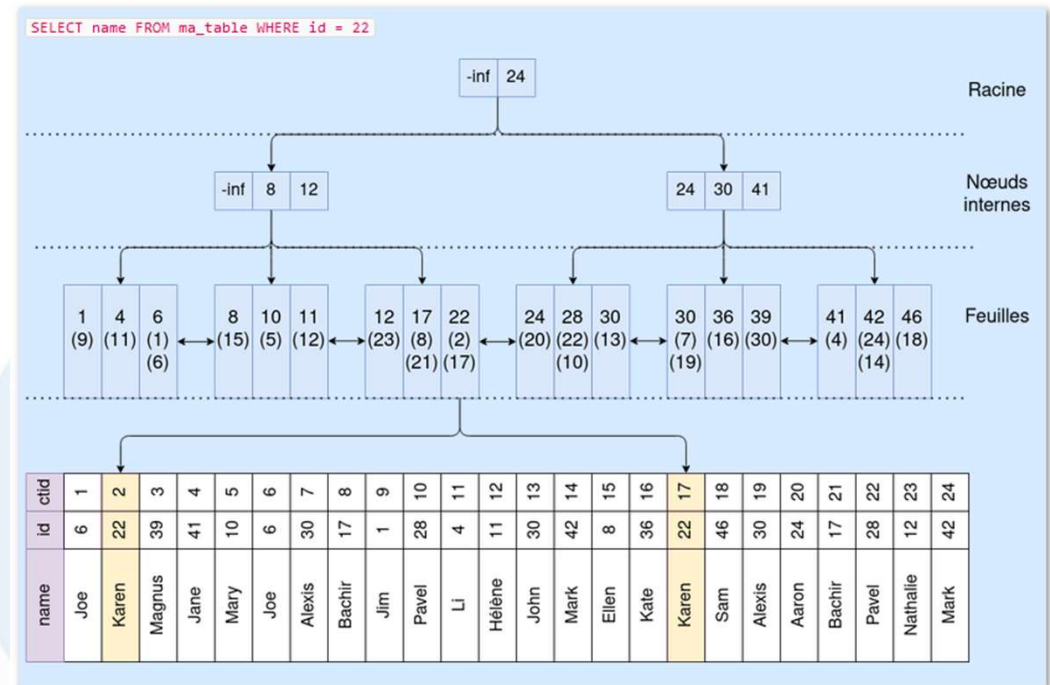
Représentation simplifiée

INDEX B-TREE

MySQL stocke les index dans une structure de type "arbre" dit **index B-tree** contenant trois types de nœuds

- **la racine** : elle est unique c'est la base de l'arbre
- **des nœuds internes** : il peut y en avoir plusieurs niveaux
- **des feuilles** : elles contiennent :
 - les **valeurs indexées** (triées !)
 - les **valeurs incluses** (si applicable)
 - les **positions physiques (ctid)**, ici entre parenthèses et sous forme abrégée, car la forme réelle est (numéro de bloc, position de la ligne dans le bloc)
 - l'adresse de la feuille précédente et de la feuille suivante.

La racine et les nœuds internes contiennent des enregistrements qui décrivent la valeur minimale de chaque bloc du niveau inférieur et leur adresse (ctid)



Ce schéma présente une vue simplifiée d'une table (en blanc, avec ses champs **id** et **name**) et d'un index B-tree sur **id** (en bleu), tel que le créerait :

```
CREATE INDEX mon_index ON ma_table (id) ;
```


EXEMPLE

Recherchons le résultat en passant par l'index de :

```
SELECT name FROM ma_table WHERE id = 22
```

- En parcourant la racine, on cherche un enregistrement dont la valeur est strictement supérieure à la valeur que l'on recherche. Ici, 22 est plus petit que 24 : on explore donc le nœud de gauche.
- Ce nœud référence trois nœuds inférieurs (ici des feuilles).
- On compare de nouveau la valeur recherchée aux différentes valeurs (triées) du nœud : pour chaque intervalle de valeur, il existe un pointeur vers un autre nœud de l'arbre.
- Ici, 22 est plus grand que 12, on explore donc le nœud de droite au niveau inférieur.
- Une fois arrivé sur une feuille, il suffit de la parcourir pour récupérer l'ensemble des positions physiques des lignes correspondants au critère.
- Ici, la feuille nous indique qu'à la valeur 22 correspondent deux lignes aux positions 2 et 17.
- Pour trouver les valeurs de name, il faut aller chercher dans la table même les lignes aux positions trouvées dans l'index.
- D'autre part, les informations de visibilité des lignes doivent aussi être trouvées dans la table.

LES INDEX

Créer un index ordinaire

```
-- La syntaxe basique pour créer un index est la suivante :  
CREATE INDEX `index_nom` ON `table`;  
  
/*  
Créer un index sur une seule colonne en précisant la colonne à indexer  
*/  
CREATE INDEX `index_nom` ON `table` (`colonne1`);  
  
-- Pour insérer un index sur plusieurs colonnes  
CREATE INDEX `index_nom` ON `table` (`colonne1`, `colonne2`);
```

Créer un index unique

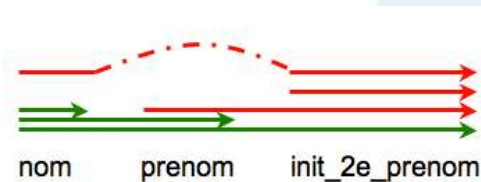
```
/*  
Un index unique permet de spécifier qu'une ou plusieurs colonnes doivent  
contenir des valeurs uniques à chaque enregistrement. Le système de base de  
données retournera une erreur si une requête tente d'insérer des données qui  
feront doublons sur la clé d'unicité.  
*/  
  
-- Pour créer ce type d'index  
CREATE UNIQUE INDEX `index_nom` ON `table` (`colonne1`);  
/*  
Dans cet exemple un index unique sera créé sur la colonne nommée colonne1. Cela  
signifie qu'il ne peut pas y avoir plusieurs fois la même valeur sur 2  
enregistrements distincts contenus dans cette table.  
*/  
  
-- Pour Créer un index d'unicité sur 2 colonnes  
CREATE UNIQUE INDEX `index_nom` ON `table` (`colonne1`, `colonne2`);
```

LES INDEX

Index sur plusieurs colonnes

Si on effectue souvent des recherches à la fois sur le nom, le prénom ? On va créer un index sur les deux colonnes.

MySQL est capable de tirer parti de votre index sur (nom, prénom) pour certaines autres recherches : "index par la gauche"



Index sur des colonnes de type alphanumérique

Types CHAR et VARCHAR : l'index décompose votre varchar caractères par caractères. On peut créer l'index à partir d'une certaine taille pour éviter de prendre la totalité de la chaîne de caractères.

DÉSAVANTAGES

On peut noter quelques désavantages de ce type d'objet en SQL.

1. Ils prennent de la place en mémoire.
2. Ils ralentissent les requêtes d'insertion, modification et suppression, puisqu'à chaque fois, il faut remettre l'index à jour en plus de la table.

Par conséquent, n'ajoutez pas d'index lorsque ce n'est pas vraiment utile.

3. Avoir un index UNIQUE sur une colonne (ou plusieurs) permet de s'assurer que jamais vous n'insérerez deux fois la même valeur (ou combinaison de valeurs) dans la table.
4. Lorsque vous mettez un index UNIQUE sur une table, vous ne mettez pas seulement un index, vous ajoutez surtout une **contrainte**.



MYSQL - TRANSACTIONS

ENCHAINEMENT DE REQUÊTES

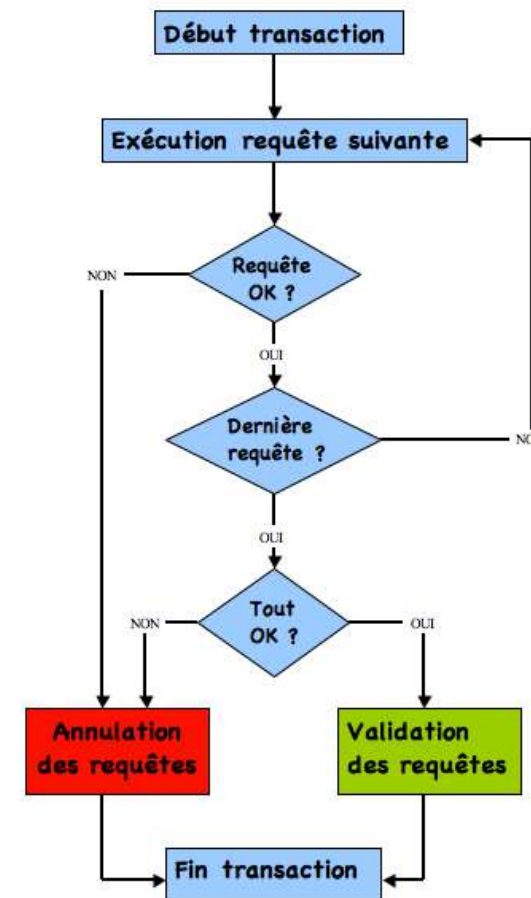
PRINCIPE

Une transaction :

- ensemble de requêtes qui sont exécutées en un seul bloc.
- si une des requêtes du bloc échoue, on peut décider d'annuler tout le bloc de requêtes (ou de quand même valider les requêtes qui ont réussi).



- *Les tables MyISAM sont non transactionnelles, donc ne supportent pas les transactions.*
- *les tables InnoDB sont transactionnelles, donc supportent les transactions.*



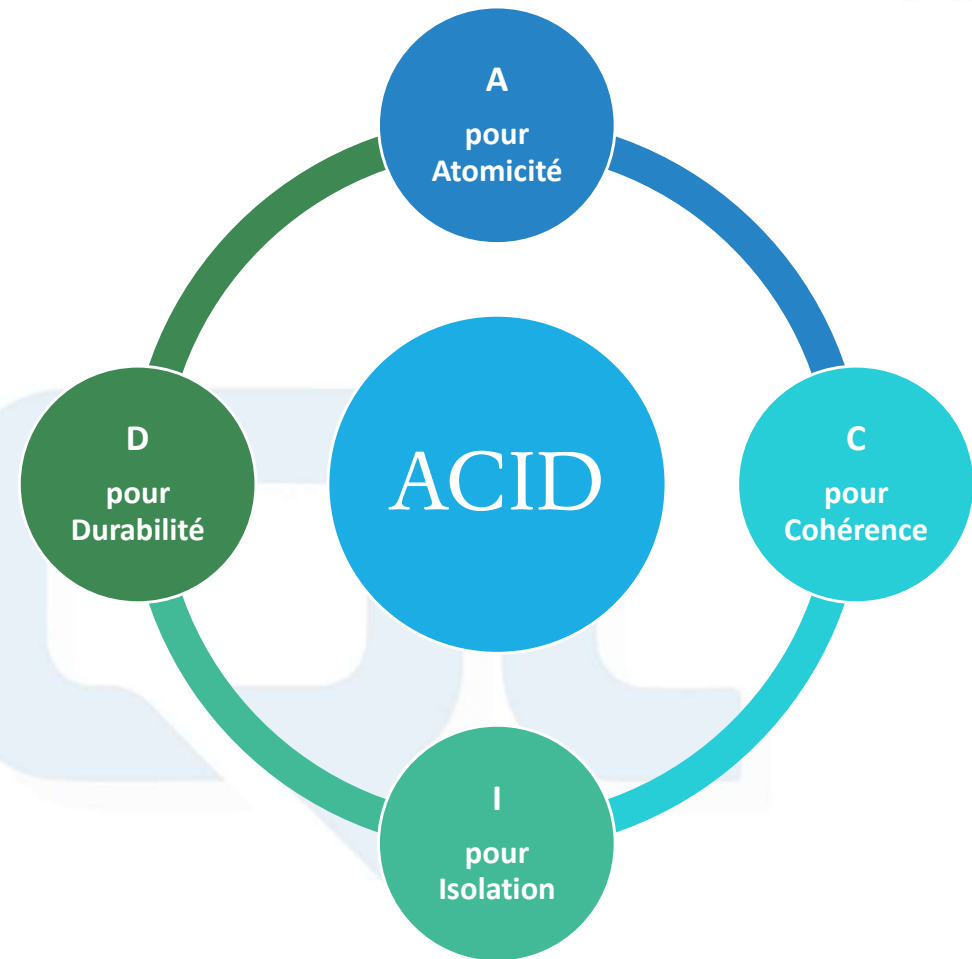
PRÉCISIONS SUR LES TRANSACTIONS

Toutes les commandes qui influent sur la structure de la base valident implicitement les transactions.

- CREATE DATABASE, DROP DATABASE ;
- CREATE TABLE, ALTER TABLE, RENAME TABLE, DROP TABLE, CREATE INDEX, DROP INDEX, CREATE PROCEDURE etc...



Les transactions doivent respecter les propriétés ACID qui garantissent qu'une transaction est exécutée de façon fiable.



PRINCIPE ACID

Atomicité

L'atomicité garantit que chaque transaction est traitée comme une seule "unité", qui réussit complètement ou échoue complètement :

- si l'une des déclarations constituant une transaction échoue, la transaction entière échoue et la base de données reste inchangée.

Cohérence

La cohérence garantit qu'une transaction ne peut faire passer la base de données que d'un état cohérent à un autre, en préservant les invariants de la base de données :

- toute donnée écrite dans la base de données doit être valide selon toutes les règles définies, y compris les contraintes, les rollbacks, les déclencheurs et toute combinaison de ceux-ci.

Cela empêche la corruption de la base de données par une transaction illégale.

PRINCIPE ACID

Isolation

L'isolation garantit que l'exécution simultanée des transactions laisse la base de données dans le même état que celui qui aurait été obtenu si les transactions avaient été exécutées séquentiellement.

Durabilité

La durabilité garantit qu'une fois qu'une transaction a été validée, elle le restera même en cas de défaillance du système (par exemple, une panne de courant ou un crash).

Cela signifie généralement que les transactions terminées (ou leurs effets) sont enregistrées dans une mémoire non volatile.

FONCTIONNEMENT DANS INNODB

Dans InnoDB, toutes les activités des utilisateurs se produisent à l'intérieur d'une transaction.

- Si autocommit est activé, chaque instruction SQL forme une seule transaction.
- Par défaut, MySQL démarre la session pour chaque nouvelle connexion avec autocommit activé, donc MySQL effectue un commit après chaque instruction SQL si cette instruction n'a pas renvoyé d'erreur.
- Si une instruction renvoie une erreur, le comportement de validation ou de restauration dépend de l'erreur.
- Si autocommit est désactivé dans une session avec `SET autocommit = 0`, la session a toujours une transaction ouverte. Une instruction `COMMIT` ou `ROLLBACK` termine la transaction en cours et une nouvelle commence.
- Un `COMMIT` signifie que les modifications apportées à la transaction en cours sont rendues permanentes et deviennent visibles pour les autres sessions.
- Un `ROLLBACK` instruction, en revanche, annule toutes les modifications apportées par la transaction en cours. Les deux `COMMIT` et `ROLLBACK` libèrent tous les verrous qui ont été définis pendant la transaction en cours.

SYNTAXE ET UTILISATION

START TRANSACTION ou **BEGIN** : démarrer une nouvelle transaction.

COMMIT valide la transaction en cours, rendant ses modifications permanentes.

ROLLBACK annule la transaction en cours, annulant ses modifications.

SET autocommit désactive ou active le mode de validation automatique par défaut pour la session en cours.



*Par défaut, MySQL s'exécute avec le mode de validation automatique activé. Cela signifie que, lorsqu'elle n'est pas autrement à l'intérieur d'une transaction, chaque instruction est atomique, comme si elle était entourée de **START TRANSACTION** et **COMMIT**.*

*Vous ne pouvez pas utiliser **ROLLBACK** pour annuler l'effet ; cependant, si une erreur se produit lors de l'exécution de l'instruction, l'instruction est annulée.*

```
/*
ce n'est valable que pour la session courante.
Or, en ouvrant une nouvelle connexion, vous avez créé une nouvelle session.
Il faut donc désactiver donc à nouveau ce mode.
*/
SET autocommit=0;

START TRANSACTION;

-- debut de nos requêtes
INSERT INTO...

/*
Une fois la transaction ouverte, les requêtes devront être validées pour
prendre effet.
Attention au fait qu'un COMMIT ou un ROLLBACK met fin automatiquement à la
transaction, donc les commandes suivantes seront à nouveau « commitées »
automatiquement si une nouvelle transaction n'est pas ouverte.
*/
COMMIT ou ROLLBACK;
```

LES JALONS

Il est possible de démarrer une transaction à l'intérieur d'une transaction.

On peut poser des jalons de transaction :

- points de repère qui permettent d'annuler toutes les requêtes exécutées depuis ce jalon, et non toutes les requêtes de la transaction.

```
SAVEPOINT nom_jalon;  
-- Crée un jalon avec comme nom "nom_jalon"  
  
ROLLBACK [WORK] TO [SAVEPOINT] nom_jalon;  
-- Annule les requêtes exécutées depuis le jalon "nom_jalon",  
-- WORK et SAVEPOINT ne sont pas obligatoires  
  
RELEASE SAVEPOINT nom_jalon;  
-- Retire le jalon "nom_jalon"  
-- (sans annuler, ni valider les requêtes faites depuis)
```

LES VEROUS

Complément indispensable des transactions, les verrous permettent de sécuriser les requêtes en bloquant ponctuellement et partiellement l'accès aux données.

2 types de verrous sont disponibles :

- Les verrous de tables
- Les verrous de lignes

```
-- Verrous de tables

LOCK TABLES nom_table [AS alias_table] [READ | WRITE] [, ...];

LOCK TABLES Espece READ, Adoption AS adopt WRITE;
-- On pose un verrou de lecture sur Espece
-- et un verrou d'écriture sur Adoption avec l'alias adopt

-- Verrous de lignes

SELECT * FROM Animal WHERE espece_id = 5 LOCK IN SHARE MODE;
/*
pose donc un verrou partagé sur les lignes de la table Animal pour lesquelles
espece_id vaut 5.
Ce verrou signifie en fait, pour les autres sessions : "Je suis en train de
lire ces données. Vous pouvez venir les lire aussi, mais pas les modifier tant
que je n'ai pas terminé."
*/

SELECT * FROM Animal WHERE espece_id = 5 FOR UPDATE;
/*
poser un verrou exclusif, on utilise FOR UPDATE à la fin de la requête SELECT.
Ce verrou signifie aux autres sessions : "Je suis en train de lire ces données
dans le but probable de faire une modification. Ne les lisez pas avant que
j'aie fini (et bien sûr, ne les modifiez pas)."
*/
```

VALIDATION IMPLICITE



Toutes les commandes qui créent, modifient, suppriment des objets dans la base de données valident implicitement les transactions.

Cela comprend donc :

- la création et suppression de bases de données : `CREATE DATABASE`, `DROP DATABASE` ;
- la création, modification, suppression de tables : `CREATE TABLE`, `ALTER TABLE`, `RENAME TABLE`, `DROP TABLE` ;
- la création, modification, suppression d'index : `CREATE INDEX`, `DROP INDEX` ;
- la création d'objets comme les procédures stockées, les vues, etc., dont nous parlerons plus tard.

La création, la modification et la suppression d'utilisateurs provoquent aussi une validation implicite.

la commande `START TRANSACTION` provoque également une validation implicite si elle est exécutée à l'intérieur d'une transaction. Le fait d'activer le mode autocommit (s'il n'était pas déjà activé) a le même effet.

La création et suppression de verrous de table clôturent aussi une transaction en la validant implicitement.



MYSQL - LES FONCTIONS

ÉCRIRE NOS PROPRES FONCTIONS

LES FONCTIONS

Définition

Une fonction :

- code qui effectue une série d'instructions bien précises et renvoie le résultat de ces instructions.
- définie par son nom et ses paramètres.
- peut avoir un ou plusieurs paramètres, ou n'en avoir aucun.
- L'ordre de ces paramètres est très important.

Lorsque l'on utilise une fonction, on dit que l'on fait appel à celle-ci avec **SELECT**

- il suffit donc de donner son nom, suivi des paramètres éventuels entre parenthèses (obligatoires, même s'il n'y a aucun paramètre).

Quelques exemples

```
-- Fonction sans paramètre
SELECT PI();      -- renvoie le nombre Pi, avec 5 décimales
-- Fonction avec un paramètre
SELECT MIN(prix) AS minimum
FROM Espece;
-- Fonction avec plusieurs paramètres
SELECT REPEAT('fort ! Trop ', 4);
-- répète une chaîne (ici : 'fort ! Trop ', répété 4 fois)
```


LES FONCTIONS D'AGRÉGATION

Définition

Les fonctions d'agrégation dans le langage SQL permettent d'effectuer des opérations statistiques sur un ensemble d'enregistrement.

Étant donné que ces fonctions s'appliquent à plusieurs lignes en même temps, elles permettent des opérations qui servent à récupérer l'enregistrement le plus petit, le plus grand ou bien encore de déterminer la valeur moyenne sur plusieurs enregistrements.

Fonctions d'agrégation statistiques

Les fonctions d'agrégation sont des fonctions idéales pour effectuer quelques statistiques de bases sur des tables.

Les principales fonctions sont les suivantes :

- AVG()
- COUNT()
- MAX()
- MIN()
- SUM()

LES FONCTIONS D'AGRÉGATION

Utilisation

```
SELECT fonction(colonne) FROM table;
```

La fonction COUNT() possède une subtilité - Pour compter le nombre total de ligne d'une table, il convient d'utiliser l'étoile * :

```
SELECT COUNT(*) FROM table
```

Utilisation avec GROUP BY

Toutes ces fonctions prennent toutes leur sens lorsqu'elles sont utilisées avec la commande GROUP BY qui permet de filtrer les données sur une ou plusieurs colonnes.

LES FONCTIONS SCALAIRES

Définition

Les fonctions scalaires s'appliquent à chaque ligne indépendamment.

Il existe trois types de fonction scalaire SQL :

- les fonctions de date/heure,
- les fonctions de conversion
- les fonctions de chaîne.

Fonctions DATE et HEURE

SQL offre toute une panoplie de fonctions pouvant convertir, manipuler les types dates et heures dont voici quelques exemples :

- DATEDIFF() : qui donne un résultat en nombre de jours.
- TIMEDIFF() : qui donne un résultat sous forme de TIME.
- TIMESTAMPDIFF() : qui donne le résultat dans l'unité de temps souhaitée (heure, secondes, mois...).
- DATEDIFF(date1, date2)
- TIMEDIFF()
- ADDDATE() : qui s'utilise avec un INTERVAL ou un nombre de jours.

FONCTIONS CHAÎNES DE CARACTÈRES

Les fonctions SQL sur les chaînes de caractères permettent d'ajouter de nombreuses fonctionnalités aux requêtes SQL.

Ces fonctions sont mono-lignes : cela signifie qu'elles ne s'appliquent qu'à une seule ligne en même temps.

- **CHAR_LENGTH()** permet de compter le nombre de caractères.
- **CONCAT()** concaténer plusieurs chaînes de caractères
- **FORMAT()**
- **LENGTH()** retourner la longueur d'une chaîne
- **LOWER()** transformer la chaîne pour tout retourner en minuscule.
- **LTRIM()** supprimer les caractères vides au début de la chaîne.
- **REPLACE()** remplacer des caractères par d'autres caractères.
- **SUBSTR()** retourne un segment de chaîne.
- **SUBSTRING()** retourne un segment de chaîne.
- **TRIM()** supprime les caractères vides en début et fin de chaîne.
- **UPPER()** tout retourner en majuscule.

FONCTIONS UTILISATEUR

Définition

Les fonctions définies par l'utilisateur ne permettent pas d'exécuter des actions qui modifient l'état des bases de données.

Elle cherche une donnée, calcule une donnée et la retourne.

Création et suppression d'une fonction

```
CREATE FUNCTION <foo> ([params..])  
  RETURNS <datatype>  
  [ [NOT] DETERMINISTIC ]  
  BEGIN  
    <code>  
  RETURN <code>  
  END  
  
DROP FUNCTION [if exists] <foo>
```

FONCTIONS UTILISATEUR

Fonctions déterministes

Les fonctions déterministes retournent toujours le même résultat chaque fois qu'elles sont appelées avec un même ensemble spécifique de valeurs d'entrée et donnent le même état de la base de données.

Par exemple, la fonction `AVG()`.

Fonctions non déterministes

Les fonctions non déterministes peuvent retourner des résultats différents chaque fois qu'ils sont appelés avec un même ensemble spécifique de valeurs d'entrée, même si l'état de base de données auquel ils accèdent reste le même.

Par exemple, la fonction `GETDATE()`.

EXEMPLE

Exemple de fonction avec deux paramètres

```
DELIMITER |
DROP FUNCTION IF EXISTS verifier_date;
CREATE FUNCTION verifier_date (jour DATETIME, heure TIME)
RETURNS VARCHAR(10) DETERMINISTIC
BEGIN
    IF (
        NOT(MONTH(jour) IN (7,8) OR
            (MONTH(jour) = 6 AND DAY(jour) >=15) OR
            (MONTH(jour) = 9 AND DAY(jour)<=15)) AND
            ((DAYOFWEEK(jour) BETWEEN 2 AND 5)
            OR (DAYOFWEEK(jour) = 6 AND heure < '19:00:00')
            OR (DAYOFWEEK(jour) = 1 AND heure > '10:00:00')))
    THEN RETURN 'Incorrect';
    ELSE
        RETURN 'Correct';
    END IF;
END |
DELIMITER ;
```

EXEMPLE

Fonction avec 1 paramètre

```
DELIMITER |
DROP FUNCTION IF EXISTS controle_rencontre;
CREATE FUNCTION controle_rencontre (NomRencontre CHAR(20))
RETURNS VARCHAR(8) DETERMINISTIC
> BEGIN
>   IF (SELECT count(*) FROM passage P
>       JOIN rencontre R
>         ON R.REN_ID = P.REN_ID
>       WHERE R.REN_NOM LIKE NomRencontre AND
>             NOT(MONTH(P.PAS_DATE) IN (7,8) OR
>             (MONTH(P.PAS_DATE) = 6 AND DAY(P.PAS_DATE) >=15) OR
>             (MONTH(P.PAS_DATE) = 9 AND DAY(P.PAS_DATE)<=15)) AND
>             ((DAYOFWEEK(P.PAS_DATE) BETWEEN 2 AND 5)
>             OR (DAYOFWEEK(P.PAS_DATE) = 6 AND P.PAS_HEUREDEB < '19:00:00')
>             OR (DAYOFWEEK(P.PAS_DATE) = 1 AND P.PAS_HEUREDEB > '10:00:00')) > 0
>       THEN RETURN 'Pas Bien';
>   ELSE
>       RETURN 'Bien';
>   END IF;
> END |
DELIMITER ;
```




MYSQL - PROCÉDURES STOCKÉES

METTRE EN PLACE DES PROCÉDURES

PROCÉDURES STOCKÉES

Définition

Une procédure stockée, (stored procedure)

- concept utilisé en administration de base de données afin d'exécuter un ensemble d'instructions SQL.
- peut être appelée à tout moment par son nom afin d'exécuter celle-ci avec l'instruction **CALL**

Intérêts

- **Simplifier** : éviter la redondance de code.
- **Amélioration des performances** :
 - exécutée côté serveur : résultat envoyé directement au client
= réduit les échanges entre le client et le serveur,
- **Sécurité** : des applications peuvent avoir accès uniquement aux procédures stockées, sans avoir accès aux données des tables directement, et/ou s'assurer que l'accès aux données soit toujours effectué de la même manière.

SYNTAXE

Comme pour les fonctions, nous avons une syntaxe identique pour la création et la destruction.



Délimiteur : par défaut ; ce qui peut poser problèmes. Donc on peut changer de délimiteur ainsi le caractère qui délimite les instructions par l'instruction **DELIMITER**.

- *Cela n'agit que sur la session courante.*

```
CREATE PROCEDURE nom_procedure ([parametre1 [, parametre2, ...]])
BEGIN
    instruction1;
    instruction2;
    ...
    instruction3;
END;

DROP PROCEDURE nom_procedure;

CALL nom_procedure(...);

-- avec changement de délimiteur
DELIMITER | -- On change le délimiteur
CREATE PROCEDURE afficher_races()
| -- toujours pas de paramètres, toujours des parenthèses
BEGIN
|     SELECT id, nom, espece_id, prix
|     FROM Race; -- Cette fois, le ; ne nous embêtera pas
END| -- Et on termine bien sûr la commande CREATE PROCEDURE par
| notre nouveau délimiteur
DELIMITER ;
```

LA GESTION DES PARAMÈTRES

Gestion des paramètres

Un paramètre peut être de trois sens différents : entrant (**IN**), sortant (**OUT**), ou les deux (**INOUT**).

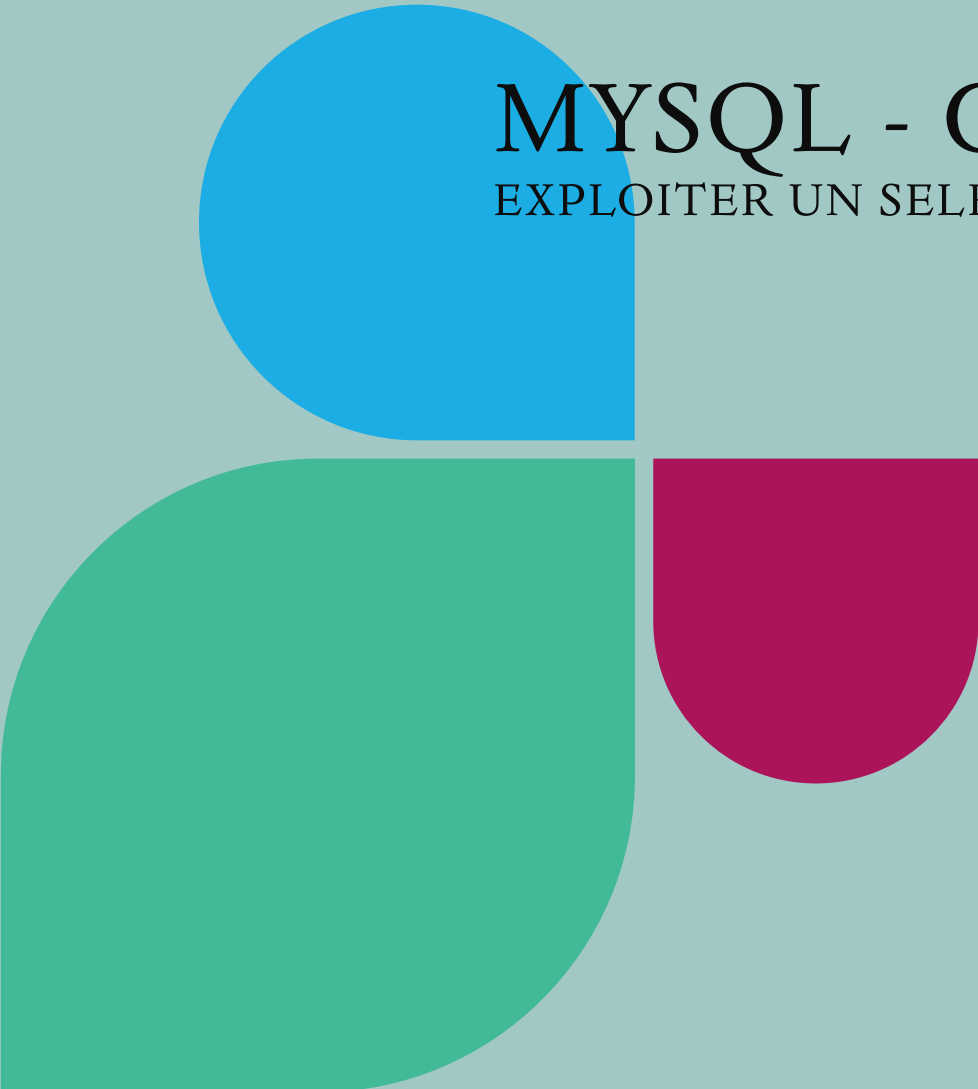
IN : c'est un paramètre "entrant". C'est-à-dire qu'il s'agit d'un paramètre dont la valeur est fournie à la procédure stockée. Cette valeur sera utilisée pendant la procédure (pour un calcul ou une sélection, par exemple).

OUT : il s'agit d'un paramètre "sortant", dont la valeur sera établie au cours de la procédure et qui pourra ensuite être utilisé en dehors de cette procédure.

INOUT : un tel paramètre sera utilisé pendant la procédure, verra éventuellement sa valeur modifiée par celle-ci, et sera ensuite utilisable en dehors.

Exemple

```
DELIMITER | -- Facultatif si votre délimiteur est toujours |
CREATE PROCEDURE afficher_race_selon_espece (IN p_espece_id INT)
| -- Définition du paramètre p_espece_id
BEGIN
| SELECT id, nom, espece_id, prix
| FROM Race
| WHERE espece_id = p_espece_id; -- Utilisation du paramètre
END |
DELIMITER ; -- On remet le délimiteur par défaut
```

A decorative graphic on the left side of the slide. It features a blue circle in the upper left, a green circle in the lower left, and a magenta circle in the lower right. A thin blue vertical line is positioned to the left of the blue circle.

MYSQL - CURSEURS

EXPLOITER UN SELECT

CURSEURS

Les curseurs permettent de parcourir un jeu de résultats d'une requête SELECT, quel que soit le nombre de lignes récupérées, et d'en exploiter les valeurs.

Quatre étapes sont nécessaires pour utiliser un curseur.

- Déclaration du curseur : avec une instruction DECLARE.
- Ouverture du curseur : on exécute la requête SELECT du curseur et on stocke le résultat dans celui-ci.
- Parcours du curseur : on parcourt une à une les lignes.
- Fermeture du curseur.

```
DELIMITER |
CREATE PROCEDURE nom_procedure()
BEGIN
    DECLARE curseur_client CURSOR FOR SELECT * FROM Client;

    -- ouverture du curseur
    OPEN curseur_client;

    -- parcours du curseur
    FETCH curseur_client INTO variables;
    /* REMARQUE
    variables correspond à autant de variables
    que l'on récupère de colonnes avec SELECT
    */
    -- fermeture du curseur
    CLOSE curseur_client;
END |
DELIMITER ;
```

PARCOURIR TOUS LES RÉSULTATS

Pour parcourir tous les résultats d'un curseur, on va pouvoir utiliser les structures de boucles telles que WHILE, REPEAT ou LOOP.

Mais il faut donc gérer la fin du parcours du curseurs soit quand il n'y a plus de données : ***ERROR 1329 (02000): No data - zero rows fetched, selected, or processed***

Pour cela, on va conditionner pour arrêter la boucle en déclarant un gestionnaire pour la condition NOT FOUND

```
DELIMITER |
CREATE PROCEDURE test_condition2(IN p_ville VARCHAR(100))
BEGIN
    DECLARE v_nom, v_prenom VARCHAR(100);
    DECLARE fin TINYINT DEFAULT 0; -- Variable locale utilisée pour stopper la boucle
    DECLARE curs_clients CURSOR
        FOR SELECT nom, prenom
        FROM Client
        WHERE ville = p_ville;

    -- Gestionnaire d'erreur pour la condition NOT FOUND
    DECLARE CONTINUE HANDLER FOR NOT FOUND SET fin = 1;

    OPEN curs_clients;
loop_curseur: LOOP
    FETCH curs_clients INTO v_nom, v_prenom;
    -- Structure IF pour quitter la boucle à la fin des résultats
    IF fin = 1 THEN
        LEAVE loop_curseur;
    END IF;

    SELECT CONCAT(v_prenom, ' ', v_nom) AS 'Client';
END LOOP;
CLOSE curs_clients;
END|
DELIMITER ;
```

A decorative graphic on the left side of the slide. It features a blue circle in the upper left, a large green rounded rectangle in the lower left, and a magenta rounded rectangle in the center. A thin blue vertical line is positioned to the left of the blue circle.

MYSQL - LES TRIGGERS

METTRE EN PLACE DES DÉCLENCHEURS

LES TRIGGERS

Définition

Le trigger

- comme une procédure mais exécutée automatiquement.
- créés pour effectuer certaines tâches en réponse à l'occurrence d'un événement spécifié (**INSERT**, **DELETE** et **UPDATE**).

À quoi sert un trigger ?

- Contraintes et vérifications de données
- Intégrité des données
- Historisation des actions
- Mise à jour d'informations qui dépendent d'autres données

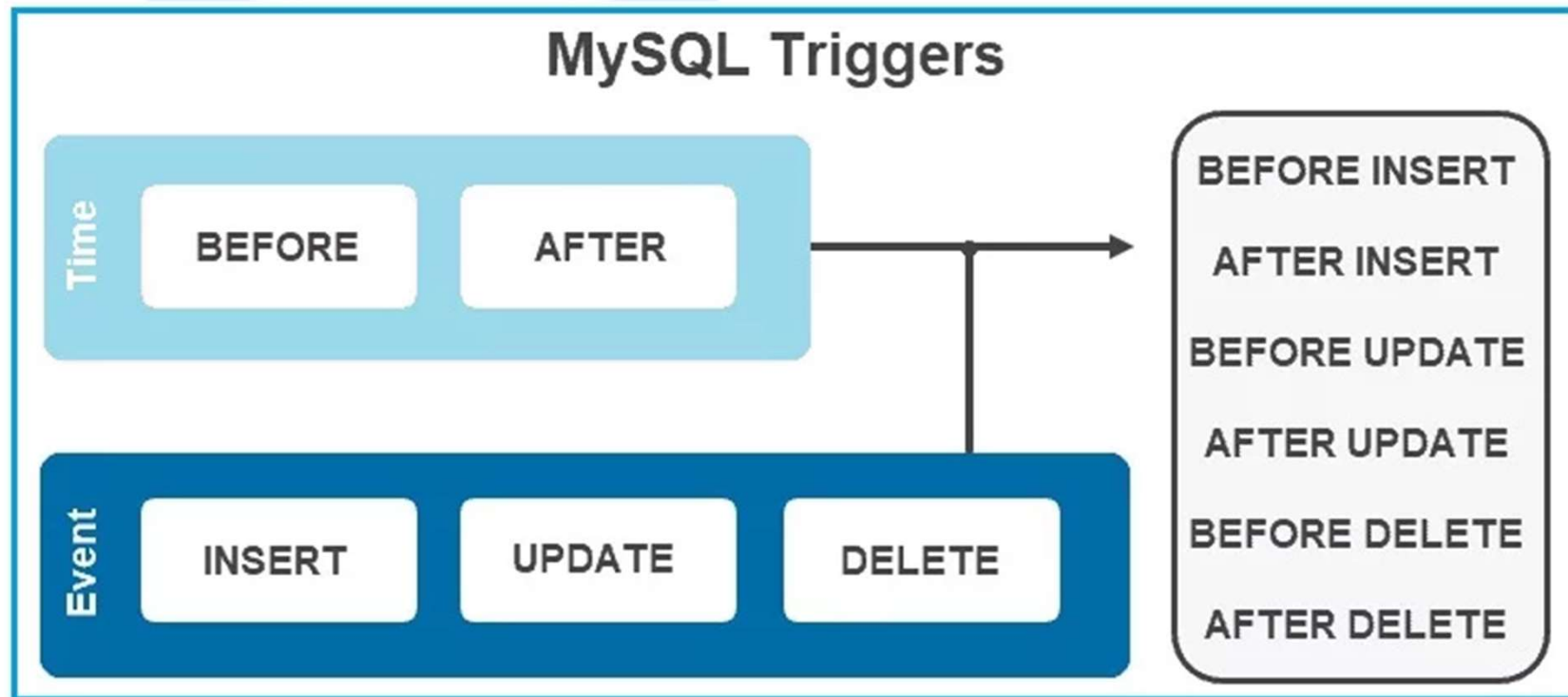
Principe

Lorsqu'un trigger est déclenché, ses instructions peuvent être exécutées à deux moments différents : soit juste avant que l'événement déclencheur n'ait lieu (**BEFORE**), soit juste après (**AFTER**).

Il ne peut exister qu'un seul trigger par combinaison [moment du trigger/événement du trigger] par table.

On a donc **un maximum de 6 triggers par table**.

RAPPEL



LES TRIGGERS

Syntaxe

```
-- syntaxe du trigger
CREATE TRIGGER nom_du_trigger
BEFORE/AFTER <events>
| CONDITION
ACTION;

-- exemple
CREATE OR REPLACE TRIGGER employee_commission_trigger
| BEFORE INSERT ON employee
| FOR EACH ROW
BEGIN
| IF NEW.deptno = 22 THEN
|   NEW.commission := NEW.salary * .2;
| END IF;
END;

DROP TRIGGER nom_du_trigger;
```

Les mots-clés OLD et NEW

Dans le corps du trigger, MySQL met à disposition deux mots-clés : **OLD** et **NEW**.

OLD représente les valeurs des colonnes de la ligne traitée avant qu'elle ne soit modifiée par l'événement déclencheur. Ces valeurs peuvent être lues, mais pas modifiées.

NEW représente les valeurs des colonnes de la ligne traitée après qu'elle a été modifiée par l'événement déclencheur. Ces valeurs peuvent être lues et modifiées.

EXEMPLE

Pour un UPDATE :

- OLD et NEW existeront

Pour un INSERT :

- OLD n'existe pas
- NEW existera

Pour un DELETE :

- OLD existera
- NEW n'existera pas

```
delimiter |
create trigger before_update_contract before update
on CONTRACT for each row
begin
    -- cas du CDD
    if old.idStatus = '2' and new.idStatus > 2 then
        signal sqlstate "45000" set message_text = "Modification du status CDD en erreur";
    -- cas du CDI
    elseif old.idStatus = '1' and new.idStatus > 1 then
        signal sqlstate "45000" set message_text = "Modification du status CDI en erreur";
    -- cas du STA
    elseif old.idStatus = '3' and new.idStatus != '3' then
        signal sqlstate "45000" set message_text = "Modification du status STA en erreur";
    end if;
end |
```



MYSQL - GESTION DES ERREURS

CONTRÔLER LES ERREURS

SIGNAL

Avec MySQL, on génère l'exception à l'aide de l'instruction `SIGNAL`.

Pour une exception personnalisée, il est de coutume d'utiliser le `SQLSTATE '45000'`.

On peut ensuite donner au `SIGNAL` un "numéro d'erreur" `MYSQL_ERRNO` qui doit être un nombre à 5 chiffres.

- Il vaut mieux éviter d'utiliser les numéros standards de MySQL et donc commencer la numérotation des exceptions personnalisées à partir de 10001.

Enfin, Le message personnalisé est donné par la variable `MESSAGE_TEXT` et ne doit pas dépasser 64 caractères sous peine d'être tronqué.

```

delimiteur |
-- déclaration d'un trigger
CREATE TRIGGER `TR_INSERT_PARTICIPANT`
BEFORE INSERT
ON `Participer`
FOR EACH ROW
BEGIN
    DECLARE CURRENT_NB INTEGER;
    DECLARE MAX_NB INTEGER;
    -- declaration de la condition d'erreur
    DECLARE TROP_DE_PARTICIPANT CONDITION FOR 45000;
    -- déclaration d'un handler permettant d'associer un évènement
    -- à déclencher lors de la propagation de l'erreur.
    DECLARE EXIT HANDLER FOR TROP_DE_PARTICIPANT SET @error = 'Trop de
participants';

    SELECT COUNT(*)
    INTO CURRENT_NB
    FROM `Participer`
    WHERE numCourse = NEW.numCourse;

    SELECT `nbMaxParticipants`
    INTO MAX_NB
    FROM `Course`
    WHERE numCourse = NEW.numCourse;

    IF (CURRENT_NB >= MAX_NB) THEN
        SIGNAL TROP_DE_PARTICIPANT;
    END IF;
END;
delimiteur ;

```



MYSQL - REGEX

UTILISATION DES REGEX AVEC SQL

UTILISATION DES REGEX

MySQL utilise l'opérateur REGEXP ou RLIKE pour la validation d'une chaîne de caractères :

```
-- Syntaxe
SELECT 'string' REGEXP 'pattern'

-- exemples
SELECT 'a' REGEXP '^[a-z]'; -- 1
SELECT 'A' REGEXP '^[a-z]'; -- 1
SELECT '1' REGEXP '^[a-z]'; -- 0
SELECT 'é' REGEXP '^[a-z]'; -- 0

-- donne les noms commençant par 'sa'
SELECT name FROM student_tbl WHERE name REGEXP '^sa';
-- donne les noms terminant par 'on'
SELECT name FROM student_tbl WHERE name REGEXP 'on$';
-- donne les noms contenant une lettre comprise entre 'b' et 'g'
-- suivi de n'importe quel caractère, suivi de la lettre 'a'
SELECT name FROM student_tbl WHERE name REGEXP '[b-g].[a]';

select REGEXP_REPLACE('hello+$û^:world', '^[a-z]', '');
-- helloworld
```