

# Formation SQL, le DDL

---

[m2iformation.fr](http://m2iformation.fr)



# Formation SQL:

# Le Data Definition Language (DDL)

# Introduction

Ce module est centré sur le **DDL** (*Data Definition Language*) ou **langage de définition de données**

Il s'agit d'un **sous-langage** du langage SQL. Il est, comme son parent, un **langage de requête**.

Il est utilisé pour **définir, modifier et supprimer** la structure des données dans une base de données relationnelle. Son rôle principal est de permettre aux utilisateurs de **spécifier la structure des tables, des vues, des index** et d'autres objets de la base de données.

# Introduction

Le langage DDL est un langage très étendu qui est à la base de la construction de nombreux schémas de notre BDD. Il va principalement s'articuler autour de **trois commandes centrales:**

**CREATE**

Création

**DROP**

Suppression

**ALTER**

Modification

# Introduction

Ces précédentes commandes vont avoir une forte influence sur les principales structures de notre BDD. **Mais de quelle structure parle-t-on ? Qu'est-il possible de définir au sein de ce langage ?:**

- Les **databases** elles-même.
- Les **tables** contenues dans ces dbs.
- Les **contraintes d'intégrité** à appliquer à nos tables. Pour garantir une cohérence à nos datas.
- Les **index** pour optimiser le tri et la recherche de données.
- Les **vues**, "tables virtuelles" qui affichent sans stocker.



# Section 1:

# Construire des databases

# Construire des databases

Nos **trois commandes centrales** vont nous permettre dans un premier temps de construire notre base de données elle-même. Il suffira de lui donner un nom pour pouvoir la créer:

```
-- Créer une base de données  
CREATE DATABASE maBaseDeDonnées  
  
-- Supprimer une base de données  
DROP DATABASE maBaseDeDonnées
```

# Convention de nommage des objets

- Ne pas dépasser 128 caractères
- Commencer par une lettre
- Être composé de lettres, de chiffres et du caractère "\_"
- Ne pas être un mot réservé du SQL (sauf entre guillemets)
- Ne pas utiliser d'accents
- Utiliser des lettres en minuscules

**Ex :** client, bdd\_ecole, date\_inscription

# Modifier une base de données

La commande **ALTER** fonctionne elle aussi sur la database. Son rôle peut être varié selon le SGBD mais elle sert principalement à **changer le jeu de caractères (character set)** et **modifier la collation** (façon dont les caractères sont triés dans votre BDD)

```
ALTER DATABASE ma_base_de_donnees  
  CHARACTER SET utf8mb4  
  COLLATE utf8mb4_unicode_ci;
```

## Vérification d'existence

Une fois qu'on a créé notre BDD, si on souhaite relancer notre script, nous allons être confronté à une erreur. En effet, la BDD existe déjà. Pour prévenir notre SGBD de ne lancer une requête que dans le cas où elle n'a pas déjà été effectuée, on peut utiliser les clauses **IF EXISTS** et **IF NOT EXISTS**

```
CREATE DATABASE IF NOT EXISTS ma_base_de_donnees  
DROP DATABASE IF EXISTS ma_base_de_données
```

*Cette commande est valable pour tous les types de structures (tables, vues...)*



# Section 2:

# Construire des tables

## Construire des tables

Une fois notre base de données définie et que nous avons spécifié que nous souhaitons travailler à l'intérieur. Le moment est venu de **concevoir des tables**.

Les commandes de **création, suppression et modifications** sont relativement proches de celles de la BDD.

Par convention, on va nommer les tables en majuscules, au singulier et sans caractères spéciaux.

**Ex:** Employee, Order, Product.

## Définition de tables

Nous allons donc utiliser la **commande CREATE TABLE** pour créer une table, puis nous allons mettre entre parenthèses **le nom des colonnes** de notre choix suivis de **leur type**.

Chaque colonne sera séparée d'une virgule

```
CREATE TABLE Product (
    name VARCHAR(50),
    quantity INT,
    price DECIMAL,
    send_date DATE
)
```

# Les types en SQL

En SQL, les types de colonnes **déterminent le type de données pouvant être stocké dans la colonne d'une table.**

Il nous faudra choisir avec précision selon si nous voulons récupérer des nombres, du texte etc...

On peut découper ces types précis en plusieurs grandes familles de types de data.

Nous allons ici parcourir ces différentes familles, mais les types peuvent être très différents d'un SGBD à l'autre.

**Référez vous donc à l'annexe de cours dédié à votre SGBD**

# Type de données numériques

Comme leur nom l'indique, ils servent à stocker des nombres. C'est un type de données très commun. En informatique, il est découpé en deux familles **distinctes**:

- **Entiers** : Ils stockent des nombres entiers sans partie décimale. Par exemple : INTEGER, INT, SMALLINT, BIGINT.
- **Décimaux** : Ils stockent des nombres avec une partie décimale, offrant une précision fixe ou variable. Par exemple : DECIMAL, NUMERIC, FLOAT, DOUBLE.

# Type de données de chaînes de caractères

Egalement très utilisé, il nous permet de regrouper des caractères pour créer **des mots et des phrases**. On définira très souvent sa **longueur maximale entre parenthèses**.

- **Chaînes de caractères fixes** : Ils stockent des chaînes de longueur fixe. Par exemple : CHAR.
- **Chaînes de caractères variables** : Ils stockent des chaînes de longueur variable. Par exemple : VARCHAR, TEXT.

## Types de données temporelles :

- **Date** : Ils stockent les valeurs de date (année, mois, jour). Ex: DATE.
- **Heure** : Ils stockent des valeurs d'heure. Par exemple : TIME.
- **Date et heure** : Ils stockent à la fois la date et l'heure. Par exemple : DATETIME, TIMESTAMP.

## Types de données binaires :

- **Blobs** : Ils stockent des données binaires de grande taille. Par exemple : BLOB, LONGBLOB.
- **Binaire variable** : Ils stockent des données binaires de longueur variable. Par exemple : VARBINARY.

## Types de données booléennes :

- **Booléens** : Ils stockent des valeurs de vérité (VRAI ou FAUX). Par exemple : BOOLEAN, BOOL, BIT.

## Types de données géographiques :

- **Géométrie** : Ils stockent des données géométriques telles que des points, des lignes, des polygones, etc. Par exemple : GEOMETRY, POINT, LINESTRING, POLYGON.

# Suppression de tables

De la même manière que pour les bases de données, nous pouvons très facilement **supprimer une table grâce à la commande DROP**, avec ou sans vérification d'existence.

```
-- Suppression classique
DROP TABLE ma_table
-- Avec vérification
DROP TABLE IF EXISTS ma_table
```

**Attention**, toute suppression de table est **immédiate** et **définitive**.  
Toutes ses données (si elle en contient) seront perdues.

# Modification de table

La modification à l'aide de la clause **ALTER** est très complète au niveau des tables:

```
-- Ajout d'une colonne à une table :  
ALTER TABLE ma_table  
ADD nouvelle_colonne INT;  
  
-- Modification du type de données d'une colonne :  
ALTER TABLE ma_table  
MODIFY colonne_existante VARCHAR(100);  
  
-- Suppression d'une colonne:  
ALTER TABLE ma_table  
DROP COLUMN colonne_a_supprimer;
```

# Renommer ses tables et ses colonnes

La clause **\*\*ALTER\*\*** nous permet également de renommer nos tables et nos colonnes:

```
-- Renommer une table  
ALTER TABLE ancien_nom_table  
RENAME TO nouveau_nom_table;  
  
-- Renommer une colonne  
ALTER TABLE ma_table  
CHANGE ancien_nom_colonne nouveau_nom_colonne type_de_donnee;
```



# Section 3:

# Appliquer des contraintes

## Appliquer des contraintes

Maintenant que nous avons conçu des tables. Il est important de leur appliquer des contraintes. Ces contraintes vont agir comme **des règles ou des restrictions** qu'on appliquerait à nos BDD et qui nous permettent un meilleur filtrage de la data qu'on reçoit.

Les contraintes sont variées, mais leur rôle est important. **Elles contribuent à assurer la cohérence, la validité et la qualité des données stockées dans la base de données.**

# Liste des contraintes de base

1. **Vérification** (`CHECK`): Permet de spécifier une condition qui doit être respectée pour chaque ligne de la table
2. **Valeur par défaut** (`DEFAULT`): Spécifie une valeur à utiliser lorsqu'aucune valeur n'est fournie lors de l'insertion de données.
3. **Non-nullité** (`NON-NULL`): Spécifie qu'une colonne ne peut pas contenir de valeurs nulles.
4. **Unique** (`UNIQUE`): Assure l'unicité des valeurs dans une colonne ou un ensemble de colonnes.

# Intégrer des contraintes à nos tables

Vous pouvez intégrer des contraintes à la création d'une table en spécifiant les contraintes directement après la définition de chaque colonne :

```
CREATE TABLE nom_table (
    colonne1 type_de_donnees contrainte,
    colonne2 type_de_donnees contrainte,
);
```

On pourra donc préciser **UNIQUE**, **NOT NULL**, **CHECK** ou **DEFAULT**. **Mais CHECK et DEFAULT nécessitent des valeurs pour être appliquées**

## Nommer ses contraintes

Une seconde syntaxe existe pour appliquer des contraintes, plus complexe mais elle nous permet de **nommer nos contraintes**.

Il faut pour cela appliquer nos contraintes à la fin de nos colonnes grâce au mot-clé **CONSTRAINT**:

```
CREATE TABLE User (
    id INT
    nom VARCHAR(50) NOT NULL,
    age INT,
    CONSTRAINT check_age CHECK (age >= 18),
);
```

## Nommer ses contraintes

- **Faciliter la compréhension du schéma** : Les autres développeurs peuvent facilement comprendre l'intention derrière la contrainte.
- **Améliorer la lisibilité du code SQL**
- **Faciliter la maintenance et le dépannage** : Lorsqu'une erreur survient en raison d'une violation de contrainte, un nom significatif peut aider à l'identifier.
- **Simplifier la gestion** : Les noms permettent de la référencer facilement lorsqu'il est nécessaire de la modifier ou de la supprimer ultérieurement.

## Les propriétés dites "clés"

En SQL, certaines propriétés importantes sont appelées des "clés". Il existe deux clés: **La clé primaire et la clé étrangère.**

- **Clé primaire:** Elle est appelée "clé" car elle est utilisée pour accéder et identifier de manière unique chaque enregistrement dans une table. C'est essentiellement la "clé" qui ouvre la porte à une ligne spécifique dans la table.
- **Clé étrangère:** Elle est appelée "clé" car elle établit une relation entre les tables en reliant les enregistrements d'une table à ceux d'une autre. Elle agit comme une "clé" qui ouvre la porte vers une autre table.

# La clé primaire

La clé primaire garantit que chaque enregistrement dans une table est **unique** et en garantit son **identification**.

Cela signifie qu'aucun enregistrement **ne peut avoir la même valeur dans la colonne définie comme clé primaire**. Elle aide à garantir l'intégrité des données en **empêchant l'insertion de doublons**. On peut la voir comme une combinaison **NOT NULL + UNIQUE**

```
CREATE TABLE Utilisateur (
    id INT PRIMARY KEY,
    nom VARCHAR(50),
);
```

# L'Auto-incrémentation/identity

Garder unique une clé primaire peut parfois être fastidieux lorsque c'est réalisé manuellement. C'est pour cela que la plupart des SGBD proposent des fonctions d'auto-incrémentation (**AUTO\_INCREMENT** en MySQL, **SERIAL** en PostgreSQL, **IDENTITY** en SQL Server.)

Lorsque vous définissez une colonne avec la propriété d'auto-incrémentation, le SGBD se charge **automatiquement** d'attribuer une valeur à cette colonne lors de l'insertion de nouvelles lignes dans la table, **garantissant ainsi l'unicité des valeurs générées.**

# La clé étrangère

Elle a un rôle bien différent, elle sert de **connexion vers une autre table de notre BDD**.

Par exemple, dans une table de commandes, la clé étrangère peut être l'identifiant de l'employé, qui fait référence à la clé primaire de la table des employés. Cela établit un lien entre chaque commande et l'employé qui l'a passée.

## La clé étrangère - Syntaxe

```
CREATE TABLE Commande (
    id INT PRIMARY KEY,
    id_client INT,
    date_commande DATE,
    FOREIGN KEY (id_client) REFERENCES Client(id)
);
```

Dans cet exemple :

- Nous créons une table Commande.
- La colonne id\_client dans la table Commande fait référence à la colonne id dans la table Client.
- Cela établit une relation entre les commandes et les clients.

# Ajout de clé à une table

Bien sûr, comme les autres contraintes, les clés peuvent également être ajoutées après la création d'une table grâce à la clause **ALTER**.

```
-- Exemple avec une table "Utilisateurs"
ALTER TABLE Utilisateurs
ADD CONSTRAINT pk_utilisateur PRIMARY KEY (id);

-- Ou avec une table "Commandes"
ALTER TABLE Commandes
ADD CONSTRAINT fk_commandes FOREIGN KEY (id_client) REFERENCES Clients(id);
```



# Section 4:

# Construire des vues

# Qu'est ce qu'une vue ?

Une vue en SQL est une **requête SQL pré-compilée** qui est stockée dans la base de données sous la forme d'une **table virtuelle**.

Les vues permettent de simplifier la complexité des requêtes en **encapsulant une logique de requête complexe dans un objet virtuel facile à utiliser**.

Elles sont utiles car elles ne **stockent rien** dans la base de données, et ne modifient pas nos tables réelles.

# Création d'une vue

Pour créer une vue (view), il suffit, comme pour la plupart des opérations en DDL, d'utiliser le mot-clé **CREATE**, ensuite on indique ce à quoi elle correspond grâce au mot-clé **AS**.

```
CREATE VIEW nom_vue AS  
SELECT colonne1, colonne2, ...  
FROM table  
WHERE condition;
```

## Avantage des vues

- **Simplification des Requêtes** : Les vues permettent d'encapsuler des requêtes complexes, ce qui simplifie le code utilisé dans les applications.
- **Sécurité des Données** : Les vues peuvent être utilisées pour restreindre l'accès aux données en limitant les accès utilisateurs.
- **Abstraction des Données** : Les vues permettent d'abstraire la structure sous-jacente des tables, ce qui facilite la modification de la structure sans avoir à modifier les applications qui utilisent la vue.
- **Réutilisation du Code** : Les vues peuvent être utilisées pour réutiliser du code SQL commun dans plusieurs requêtes, ce qui réduit la duplication du code.

## Utilisation courante des vues

- **Sélection de Données** : Les vues peuvent être utilisées pour sélectionner des données à partir d'une ou plusieurs tables de manière simple et efficace.
- **Modification de Données** : Dans certains systèmes de gestion de base de données, il est possible de modifier les données à travers une vue, bien que cela dépende des fonctionnalités spécifiques du SGBD.
- **Jointures** : Les vues peuvent être utilisées pour simplifier les opérations de jointure en encapsulant la logique de jointure dans une vue.

# Suppression de vues

La commande **DROP VIEW** nous permet de supprimer une vue lorsqu'elle ne nous est plus utile.

```
DROP VIEW nom_vue;
```

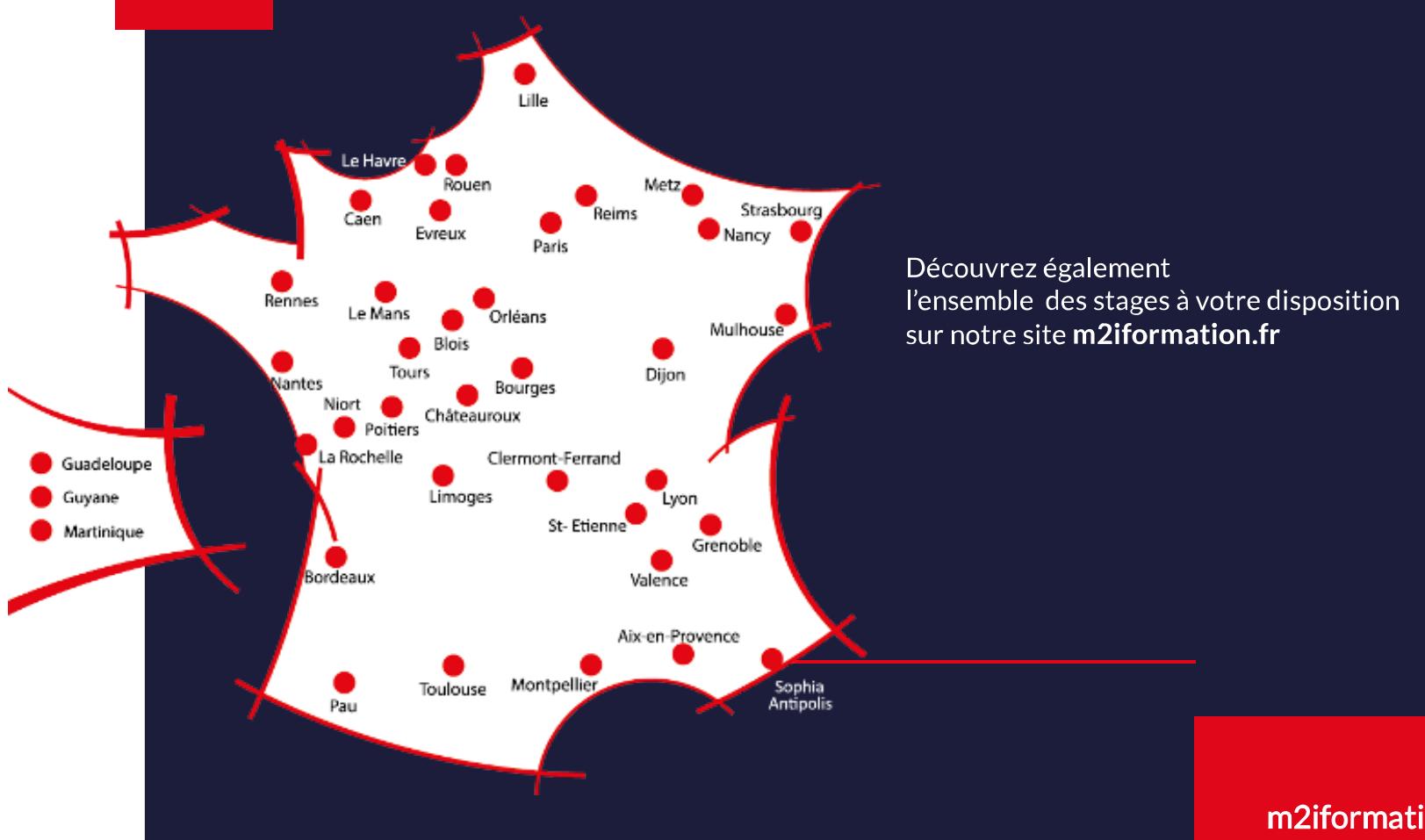
C'est une manipulation importante car

- Les vues consomment des ressources lorsqu'elles sont créées et stockées en mémoire.
- Les vues peuvent être utilisées pour accéder aux données sensibles d'une BDD. Si des vues inutilisées existent et ne sont pas correctement sécurisées, cela peut représenter un risque en plus.



**Merci pour votre attention**

**Des questions ?**



[m2iformation.fr](http://m2iformation.fr)

