SQL - Initial

Apprendre les outils de base de SQL

Date de création : 27/03/2025 - Version 1
Date de modification :

Table des matières

masolutionformation.com 101 Boulevard de l'Europe Immeuble Europa Rdc b 76100 Rouen Tel : 02 27 08 44 58 - NDA 28 27 019 7627 - Siret 799 370 200 000 39

Version 2 date de modification : 05/12/2024

Introduction

Qu'est-ce qu'une base de données ?

Une base de données est un ensemble d'informations qui est organisé de manière à être facilement accessible, géré et mis à jour. Elle est utilisée par les organisations comme méthode de stockage, de gestion et de récupération de l'information.

Les données sont organisées en lignes, colonnes et tableaux et sont indexées pour faciliter la recherche d'informations. Les données sont mises à jour, complétées ou encore supprimées au fur et à mesure que de nouvelles informations sont ajoutées. Elles contiennent généralement des agrégations d'enregistrements ou de fichiers de données, tels que les transactions de vente, les catalogues et inventaires de produits et les profils de clients.

On peut voir la base de données comme un grand fichier excel accessible par plusieurs personnes et facilement modifiable.

Pourquoi choisir une base de données plutôt qu'un fichier excel ?

Bien qu'Excel soit pratique pour de petites quantités de données, une base de données (comme MySQL ou PostgreSQL) est bien plus puissante dès que les données deviennent volumineuses ou complexes. Une BDD permet de stocker et organiser efficacement des millions d'enregistrements, tout en garantissant leur intégrité grâce à des contraintes (clés primaires, relations, etc.). De plus, SQL offre une grande flexibilité pour interroger, filtrer et manipuler les données avec des requêtes optimisées, bien plus performantes qu'Excel. Enfin, une BDD supporte plusieurs utilisateurs simultanément et sécurise mieux les accès, ce qui est essentiel pour des applications professionnelles.

Bases de données relationnelles (SQL) vs. bases de données NoSQL

Les bases de données relationnelles (comme MySQL ou PostgreSQL) organisent les données sous forme de tables liées entre elles, garantissant une structure rigoureuse et des transactions fiables (ACID). Elles sont idéales pour des données bien définies et des requêtes complexes.

À l'inverse, le NoSQL (Not Only SQL) (MongoDB, Cassandra, etc.) privilégie la flexibilité et la scalabilité horizontale*, en stockant des données sous forme de documents, graphes ou paires clé-valeur. Ces bases sont adaptées aux données non structurées (réseaux sociaux, IoT) ou aux applications nécessitant une grande vitesse d'écriture.

* *Def :* La scalabilité horizontale, ou évolutivité horizontale : elle consiste à ajouter des composants matériels pour satisfaire la demande. Il s'agit, par exemple, de s'équiper de plus de serveurs (même de manière temporaire) afin de faire face à une forte augmentation des flux et répartir les charges.

Rôle de la base de données dans le développement WEB

Les bases de données permettent aux sites Web dynamiques de récupérer, d'insérer, de mettre à jour et de supprimer des données de manière efficace, facilitant ainsi l'interaction et la personnalisation en temps réel en fonction des entrées et du comportement des utilisateurs.

Comment fonctionne une base de données sur un site WEB ?

1. L'utilisateur (vous) envoie une requête (HTTP) au serveur, via une URL. (exemple : www.google.com)
2. Le serveur récupère les données à l'intérieur de la requête effectue un traitement.
3. Si besoin le serveur demande des données à la base de données.
4. Le serveur répond à votre requête en vous envoyant une page HTML et les éléments qui y sont attachés (exemple : image, css, javascript, etc...)
5. 4. Le navigateur de l'utilisateur (toujours vous) affiche la page HTML reçu, et exécute le css/javascript s'il y en a.

Conception

Pour tout nouveau projet, vous passerez systématiquement par une phase de conception, c'est-à-dire planifier la structure du code et définir à l'avance le format de vos données. Le but ici est de concevoir un ensemble de tables liées entre elles par des relations.

Concevoir des tables

Def : Dans les bases de données relationnelles, une table est un ensemble de données organisées sous forme d'un tableau où les colonnes correspondent à des catégories d'information (une colonne peut stocker des numéros de téléphone, une autre des noms...) et les lignes à des enregistrements, également appelés entrées.

Une table est une entité métier organisée en colonnes et en lignes.

- Colonne : Champs d'information (ex : email, prix)
- Ligne : Enregistrement de l'information (ex : masolutionformation@example.org, 13.99)

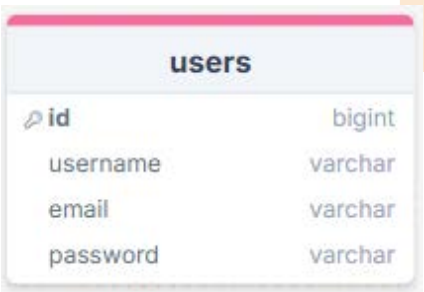
Une table doit posséder un nom avec du sens représentant ce qu'elle va stocker.
Une table possède au minimum deux colonnes (sauf certains cas, nous verrons cela ultérieurement):

- Un identifiant (ou id) représenté soit par une valeur numérique ou textuelle, **unique**. (c-à-d que chaque ligne de ma table aura un identifiant différent cela permet à la BDD d'identifier une entrée dans notre table)
- Une colonne de données (quelconque)

Règles d'or pour concevoir une bonne BDD :

- Identifier les entités principales de votre projet (ex : clients, produits, commandes, etc..) et leurs relations
- Utiliser des clés primaires (généralement l'identifiant ou l'id) pour identifier chaque ligne de manière unique.
- Utiliser des clefs étrangères pour lier les tables entre elles.
- Éviter la redondance de données. (ne stocker pas les mêmes informations à plusieurs endroits)
- Les données de chaque colonne doivent être atomique, c'est à dire qu'on ne peut plus séparer quoique ce soit. (= une information indivisible)

  Exemple : une adresse n'est pas une information atomique, mais un numéro de rue, oui , donc pour stocker une adresse il est nécessaire de la décomposer en plusieurs sous parties : numéro de rue, type de rue, nom de rue, code postal (une adresse peut donc être stockée dans un minimum de 4 colonnes.



- Pour cette session nous allons utiliser l'outil draw sql afin de faire des schémas simples dans un premier temps.
- **Contexte d'apprentissage :** nous allons réaliser la conception d'une bdd pour un mini réseau social.

Dans cet exemple nous décrivons les données que l'on souhaite stocker pour un utilisateur, chaque utilisateur enregistré sur notre base de données aura donc ce format.

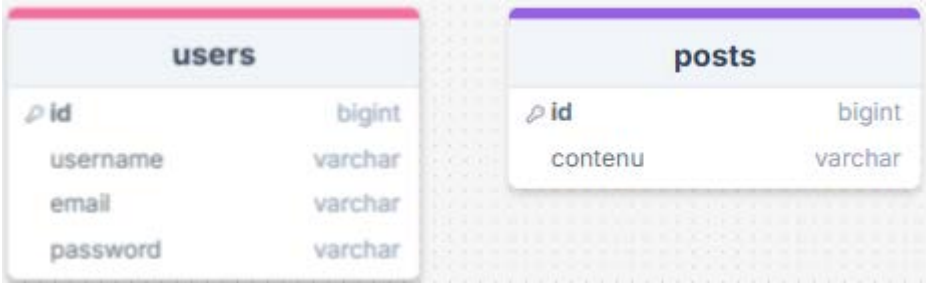
Type MYSQL vu dans l'exemple :

- bigint : valeur numérique pouvant aller de -9.223.372.036.854.775.808 à 9.223.372.036.854.775.807
- varchar : valeur textuelle

Note : Vous pouvez définir dès l'étape de conception si vous autorisez une colonne à avoir des valeurs NULL c-à-d vide.

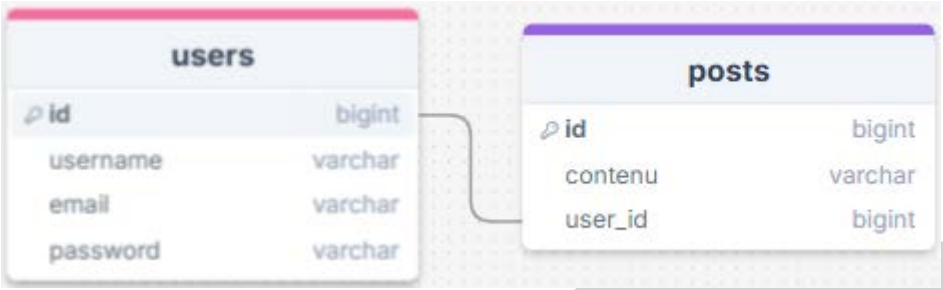
Concevoir une relation entre deux tables

Pour créer une relation entre deux tables nous allons utiliser une clef étrangère.
Reprenons notre exemple de mini réseau social :



Je souhaite créer un lien entre ma table utilisateur et ma table posts afin de déterminer qui est l'auteur d'un post.
La solution est la clé étrangère. (ou foreign key en anglais)

Def : Une clé étrangère est une colonne qui fait référence à la clé primaire d'une autre table.



Nous avons rajouté une colonne «user_id» faisant référence à la clé primaire «id» de la table users. Cela va créer un lien entre les deux tables. Et rajouter ce qu'on nomme une contrainte d'intégrité.

Cette contrainte d'intégrité dans notre exemple permettra lors de l'ajout d'un nouveau post d'empêcher de mettre un auteur de post (ici un user) qui n'existe pas. Et permet également d'empêcher la suppression d'un utilisateur qui a encore des posts reliés à lui.

On peut nommer ce genre de relation une relation « one to many » (un vers plusieurs en français) car dans cet exemple un post ne peut être écrit que par un utilisateur, et un utilisateur peut posséder entre 0 et une infinité de posts.

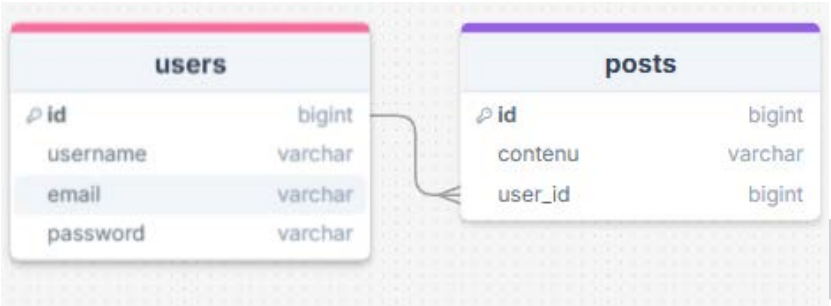
Les cardinalités

Les cardinalités définissent combien d'occurrences d'une entité peuvent être liées à une autre. Comme un feu tricolore qui régule la circulation, elles garantissent l'intégrité des données en précisant :

1:1 (Un à Un) : Une carte bancaire -> Un compte (exclusivité)

1:N (Un à Plusieurs) : Un utilisateur -> Plusieurs posts (relation classique)

N:N (Plusieurs à Plusieurs) : Étudiants -> Cours (nécessite une table intermédiaire)



Note sur la notation des relations :

Dans ce schéma, la relation entre les tables est représentée par :

Un côté «patte de corbeau» : Symbolise la multiplicité (plusieurs posts)

Un côté plat : Représente l'entité parente (1 utilisateur)

Cette notation indique qu'un utilisateur (parent) peut être associé à plusieurs posts (enfants) - une relation «un-à-plusieurs» classique.

Relation entre deux tables avec une table intermédiaire

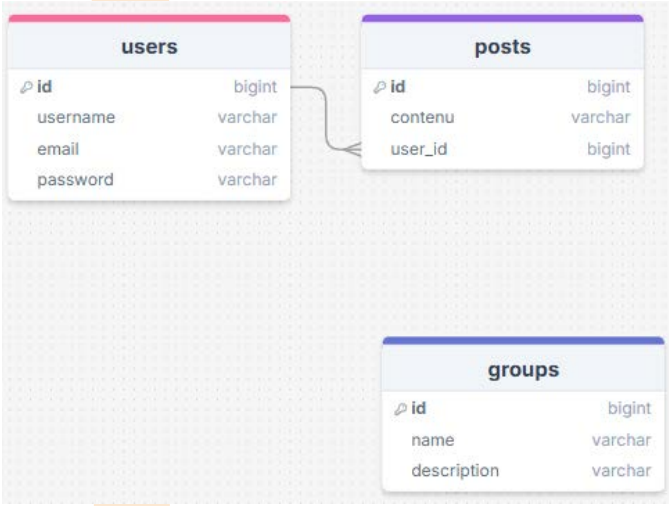
Pourquoi la relation 1-N ne suffit pas ?

Imaginons notre réseau social avec une nouvelle fonctionnalité, la fonctionnalité groupe :

Un utilisateur peut appartenir à plusieurs groupes

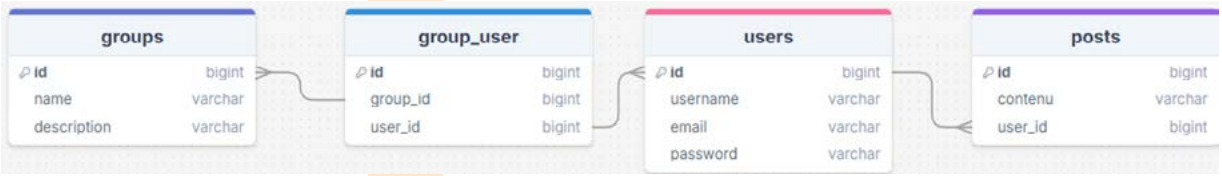
Un groupe peut contenir plusieurs utilisateurs

C'est le cas typique d'une relation plusieurs-à-plusieurs (many-to-many) qu'une simple clé étrangère ne peut pas gérer seule.



Comment relier les utilisateurs aux groupes ?

Réponse : avec une table intermédiaire nommée group_user !



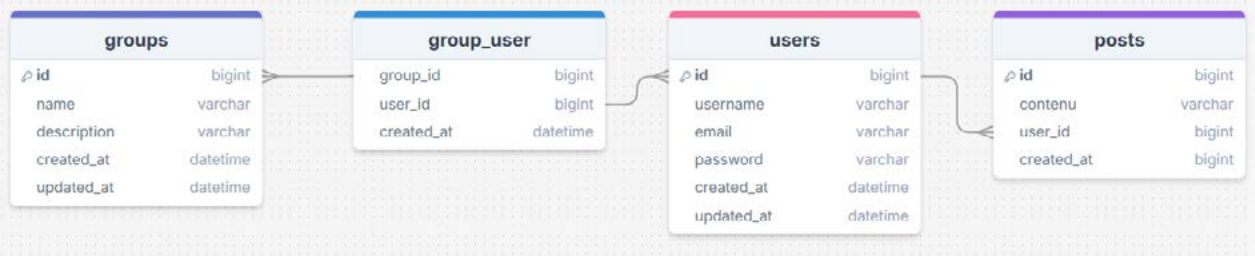
Cette table joue un rôle essentiel de répertoire des membres, enregistrant systématiquement toutes les associations entre utilisateurs et groupes dans notre mini réseau social.

Particularité technique :

- La colonne id de la table group_user est optionnelle ici
- La combinaison (group_id, user_id) forme une clé naturelle unique car :
  - Un utilisateur ne peut rejoindre un même groupe qu'une seule fois

Cette paire de valeurs identifie parfaitement chaque adhésion.

Quelques améliorations



Pourquoi est-ce une bonne idée d'ajouter des colonnes created_at (créée le) et updated_at (mis à jour le) à vos tables ?

Dans group_user

created_at :

- Enregistre la date/heure exacte où un utilisateur a rejoint un groupe
- Exemple : Savoir qui a intégré le groupe «Développeurs PHP» en premier

Dans users

created_at :

- Marque l'inscription de l'utilisateur
- Utile pour : Fêter les «1 an» d'un membre

updated_at :

- Trace les modifications du profil (photo, bio...)
- Utile pour : Détecter les comptes inactifs (si pas modifié depuis 6 mois)

Dans posts

created_at :

- Fixe la publication originale
- Utile pour : Trier les posts du plus récent au plus ancien

Dans groups

created_at :

- Conserve la création du groupe

updated_at :

- Signale les changements (nom, description...)
- Utile pour : Vérifier si un groupe est encore actif

Normes

En tant que développeur, vous travaillerez toujours en équipe. Pour garantir une collaboration efficace, il est crucial d'adopter des conventions de nommage claires pour vos tables et colonnes.

1. **Langue**
   - Utilisez exclusivement l'anglais pour tous les noms (tables, colonnes, etc.)
2. **Format**
   - **Adoptez le snake_case :**
     - Tout en minuscules
     - Espaces remplacés par des underscores «_»
     - Exemple : user_profile au lieu de UserProfile
3. **Nommage des Tables**
   - **Entités principales : utilisez le pluriel**
     - users au lieu de user
     - products au lieu de product
   - **Tables d'association (relations many-to-many) :**
     - Combinez les noms des deux tables au singulier
     - Par ordre alphabétique
     - Exemple : group_user (pas user_group)
4. **Clés Étrangères**

- Structure : nom_table_singulier_id
- Exemple : Dans une table posts, la clé vers users sera user_id

Version 2 date de modification : 05/12/2024

Exercice pratique ! Aller voir le fichier consigne_ep_1.md

SQL

Présentation

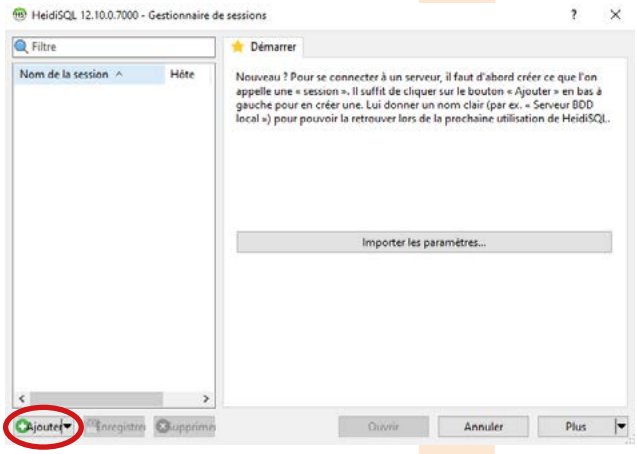
SQL ou structured query language est un langage de programmation pour base de données relationnelle.
SQL permet de :

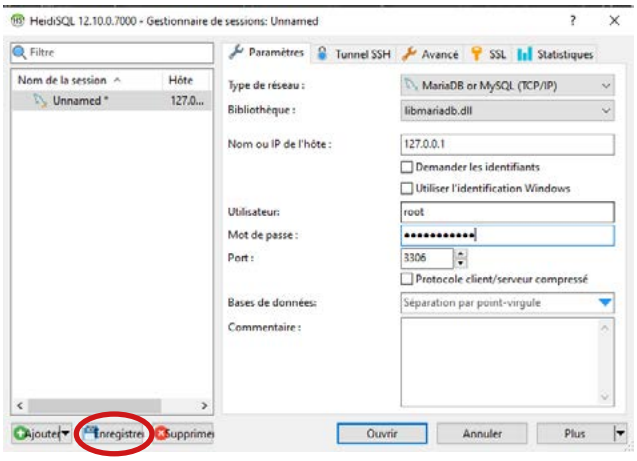
1. Définir des données (DDL - Data Definition Language) : créer/modifier/supprimer des tables dans une base de données relationnelle.
2. Manipuler des données : (DML - Data Manipulation Language) : sélectionner/insérer/modifier/supprimer des données dans une table.
3. Contrôler des données : (DCL - Data Control Language) : définir des permissions au niveau des utilisateurs d'une base de données.

Logiciel

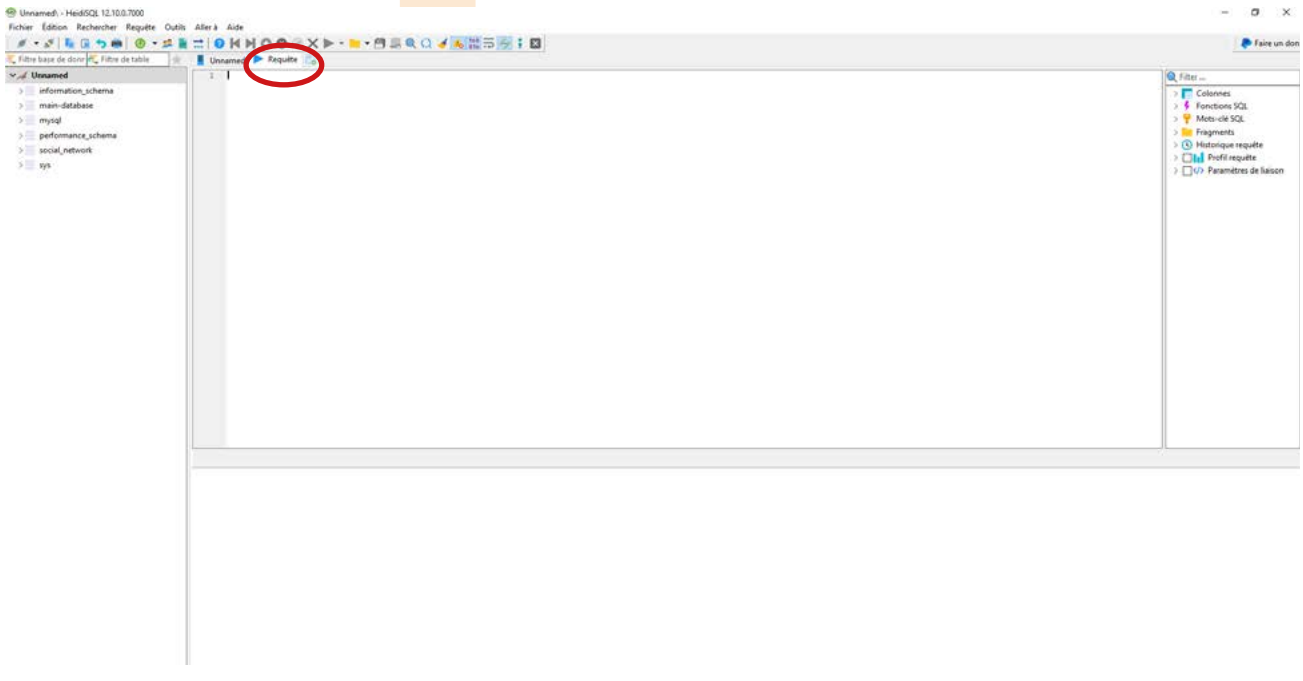
Pour cette session, nous allons utiliser le logiciel HeidiSQL.

Connexion au Serveur SQL



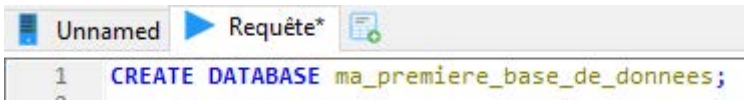


Une fois que vous avez rentré les bons identifiants et enregistré le tout, vous pouvez vous connecter avec le bouton «ouvrir».



L'onglet dans lequel nous travaillerons le plus dans cette session sera l'onglet requête.

Créer une base de données



Pour initialiser une base de données, on utilise la commande CREATE DATABASE suivie du nom choisi, terminée par un point-virgule. Cependant, dans une approche professionnelle, il est crucial d'ajouter des paramètres complémentaires pour garantir le bon fonctionnement linguistique.

```
CREATE DATABASE mini_reseau_social CHARACTER SET utf8mb4 COLLATE utf8mb4_unicode_ci;
```

Lorsque vous créez une base de données, deux paramètres sont essentiels pour gérer correctement le texte :

CHARACTER SET utf8mb4

C'est le «dictionnaire» qui définit quels caractères peuvent être stockés
Utilité :

- Supporte tous les caractères Unicode
- Permet d'utiliser des emojis, idéogrammes , lettres accentuées (éà)
- Évite les problèmes d'affichage («?» à la place des caractères spéciaux)

COLLATE utf8mb4_unicode_ci

C'est le «mode d'emploi» pour comparer et trier le texte
Utilité :

- ci = case insensitive : 'Hello' = 'HELLO'
- unicode = règles de tri intelligentes (ex: 'é' proche de 'e')
- Détermine l'ordre alphabétique dans les résultats

Créer une table

```
DROP TABLE IF EXISTS users;
CREATE TABLE users (
   id INTEGER PRIMARY KEY AUTO_INCREMENT,
   username VARCHAR(150) NOT NULL UNIQUE,
   email VARCHAR(255) NOT NULL UNIQUE,
   password VARCHAR(255) NOT NULL,
   created_at TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP,
   updated_at TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP
);
```

À partir de cette instruction qui crée la table users, nous allons apprendre à créer des tables en SQL.

DROP TABLE IF EXISTS users

- Type : Instruction de suppression
- Fonction : Supprime la table users si elle existe déjà
- Utilité : Évite les erreurs si la table est recréée plusieurs fois

CREATE TABLE users

- Type : Déclaration de création
- Fonction : Débute la définition d'une nouvelle table nommée users

id INTEGER PRIMARY KEY AUTO_INCREMENT

- Type : Colonne numérique (entier)
- Taille : Pas besoin de précision pour INTEGER
- Nullable : Non (PRIMARY KEY implique NOT NULL)
- Valeur par défaut : Auto-incrémentée (1, 2, 3...)
- Rôle : Identifiant unique de chaque utilisateur

username VARCHAR(150) NOT NULL UNIQUE

- Type : Texte de longueur variable
- Taille : Limité à 150 caractères
- Nullable : Non (NOT NULL)
- Contrainte : UNIQUE (interdit les doublons)
- Rôle : Stocke le pseudonyme de l'utilisateur

email VARCHAR(255) NOT NULL UNIQUE

- Type : Texte de longueur variable
- Taille : Limité à 255 caractères (standard pour les emails)
- Nullable : Non
- Contrainte : UNIQUE (un email = un compte)
- Rôle : Stocke l'adresse email

password VARCHAR(255) NOT NULL

- Type : Texte de longueur variable
- Taille : 255 caractères (pour stocker un hash sécurisé)
- Nullable : Non
- Rôle : Stocke le mot de passe chiffré

created_at TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP

- Type : Date et heure
- Nullable : Non
- Valeur par défaut : Date/heure actuelle automatique
- Rôle : Horodatage de la création du compte

updated_at TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP

- Type : Date et heure
- Nullable : Non
- Valeur par défaut : Date/heure actuelle (mais pas de ON UPDATE ici)
- Rôle : Horodatage de la dernière modification

Note : Il est important de respecter l'ordre des instructions lors de la définition de colonne sans quoi, vous aurez une erreur de syntaxe SQL et la requête ne s'exécutera pas.

Syntaxe création de colonne :

1. **Nom de la colonne**

- Règle :
  - Utiliser des noms clairs et descriptifs (username, birth_date)
  - Préférer le snake_case (minuscules avec underscores)
  - Éviter les espaces et caractères spéciaux

2. **Type de donnée**

- Exemples courants :
  - INT / BIGINT : Nombres entiers
  - VARCHAR(n) : Texte (avec n = nombre max de caractères)
  - DECIMAL(p,s) : Nombres à virgule (p = précision, s = échelle)
  - DATE / TIMESTAMP : Dates et heures
  - BOOLEAN : Vrai/Faux

3. **Taille (si applicable)**

- Obligatoire pour :
  - VARCHAR(255) : Texte de max 255 caractères
  - DECIMAL(10,2) : Nombre avec 10 chiffres max, dont 2 après la virgule
- Optionnel pour :
  - INT (taille par défaut)

4. **PRIMARY KEY (si clé primaire)**

Caractéristiques :

- Identifie chaque ligne de manière unique
- Une seule colonne par table (ou combinaison de colonnes)
- Implique automatiquement NOT NULL

5. **AUTO_INCREMENT (si nécessaire)**

- Fonctionne avec :
  - Colonnes numériques (INT, BIGINT)
- Comportement :
  - Génère automatiquement des valeurs uniques (+1 à chaque insertion)
- Souvent utilisé pour les id

6. **NOT NULL (si valeur obligatoire)**

- À utiliser :
  - Pour les champs obligatoires (email, mot de passe)
- À éviter :
  - Pour les données optionnelles (bio, numéro de téléphone)

7. **Autres contraintes optionnelles**

- UNIQUE : Interdit les doublons (ex: email)
- DEFAULT valeur : Valeur par défaut si non spécifiée
- CHECK (condition) : Validation personnalisée (ex: age > 0)

Insérer des données

```
INSERT INTO users (username, email, PASSWORD) VALUES ('alex-1', 'alexis.1.msg@gmail.com', '$2y$10$941AoVHhR/wGZdI5aSdb6.WjJdKrC9g47fuwjXZe1TiQ3R/nrdEEa');
```

Syntaxe :

1. **Initialisation de la commande**

- Utiliser les mots-clés INSERT INTO pour indiquer le début d'une opération d'insertion
- Suivi du nom de la table concernée (ex: users, products)

2. **Spécification des colonnes**

- Lister entre parenthèses les noms des colonnes à remplir
- Exemple : (username, email, password)
- Ordre important : Doit correspondre à l'ordre des valeurs fournies ensuite

3. **Déclaration des valeurs**

- Utiliser le mot-clé VALUES pour introduire les données à insérer
- Placer les valeurs entre parenthèses, dans le même ordre que les colonnes
- Exemple : Pour (username, email, password), écrire ('Alice', 'alice@ex.com', 'abc123')

4. **Règles à respecter**

- Valeurs obligatoires : Les colonnes définies comme NOT NULL doivent absolument recevoir une valeur
- Correspondance de types : Les données doivent correspondre au type déclaré de la colonne (texte, nombre, date...)
- Ordre cohérent : L'alignement colonnes/valeurs est critique pour éviter des erreurs

Note : Comme nous avons bien fait notre travail à la création de la table, nous n'avons pas besoin de renseigner ni l'id, ni created_at, et ni udapted_at car ils vont prendre une varleur par défaut.

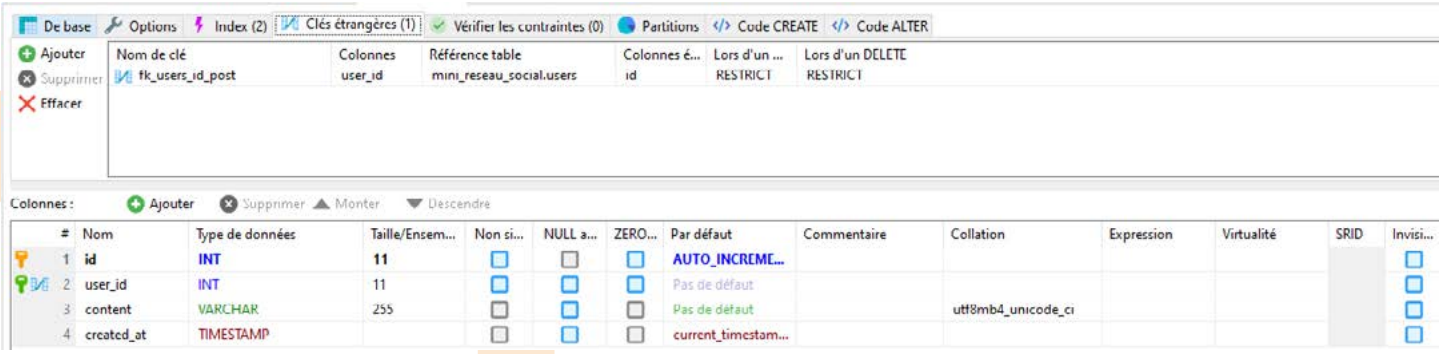
```
INSERT INTO users (username, email, PASSWORD) VALUES
   ('alex-1', 'alexis.1.msg@gmail.com', '$2y$10$941AoVHhR/wGZdI5aSdb6.WjJdKrC9g47fuwjXZe1TiQ3R/nrdEEa'),
   ('miguel', 'migoumac@yahoo.com', '$2y$10$x.Sysj/rJCzFJyg9CKZTEOft1GDaXwkTiY1y.0o30tSSwcGYLMfmq'),
   ('maya-labeille', 'mireille-danse@msn.fr', '$2y$10$wUCQTkZzmWjW9isQVVB.E.GpU.C4c1vA2QFhTN9OOXwF78vJZ1O4C');
```

L'opération INSERT INTO permet d'ajouter plusieurs enregistrements simultanément en séparant les groupes de valeurs par des virgules. Cela optimise l'exécution en réduisant les allers-retours avec la base de données.

Créer une relation one to many

Toujours avec notre projet de mini réseau social, nous allons créer la table posts représentant les posts qu'un utilisateur peut faire. Et la relier à la table users, créeant ansi une contrainte d'intégrité.

```sql
DROP TABLE IF EXISTS posts;
CREATE TABLE posts (
   id INTEGER PRIMARY KEY AUTO_INCREMENT,
   user_id INTEGER NOT NULL,
   content VARCHAR(255) NOT NULL,
   created_at TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP,
   CONSTRAINT fk_users_id_post FOREIGN KEY (user_id) REFERENCES users(id)
);
```



L'une des façons d'ajouter une clé étrangère dans une table est de le faire lors de la création de cette dernière.
Pour définir une clé étrangère directement lors de la création d'une table, deux étapes sont nécessaires :

1. **Création de la Colonne Porteuse**
- Nommez-la selon la convention <nom_table_étrangère>_id (ex: user_id pour une référence à users.id)
- Attribuez-lui le même type de données que la clé primaire référencée
2. **Déclaration de la Contrainte**
- Utilisez le mot-clé CONSTRAINT suivi d'un nom explicite (ex: fk_posts_user_id)
- Spécifiez FOREIGN KEY et indiquez entre parenthèses la colonne locale concernée
- Liez-la à la table distante via REFERENCES suivi de :
- La table cible
- Sa colonne référencée (généralement l'id)
- Cette méthode garantit l'intégrité référentielle dès la création de la structure.

Points Clés à Retenir :
- La colonne et la contrainte sont indissociables
- Le nom de la contrainte doit être compréhensible pour faciliter la maintenance
- Les types de données doivent strictement correspondre entre les deux colonnes

Insérer des données avec une clé étrangère

#	id	username	email	password	created_at	updated_at
1	1	alex-l	alexis.l.msg@gmail.com	$2y$10$941AoVHhR/wGZdI5aSdb6.WjJdKrC9g47fuwjXZ...	2025-03-27 15:12:05	2025-03-27 15:12:05
2	2	miguel	migoumac@yahoo.com	$2y$10$x.Sysj/rJCzFJyg9CKZTEOftlGDaXwkTiYly.0o30tS...	2025-03-27 15:12:05	2025-03-27 15:12:05
3	3	maya-labeille	mireille-danse@msn.fr	$2y$10$wUCQTkZzmWjW9isQVVB.E.GpU.C4c1vA2QFhT...	2025-03-27 15:12:05	2025-03-27 15:12:05
4	4	tibone	tibtib76@hotmail.fr	$2y$10$KEWDQUGrhOJlrp8opunTz.VwrsUB2Rn9FWLKdU...	2025-03-27 15:46:11	2025-03-27 15:46:11

```sql
INSERT INTO posts (user_id, content) VALUES (99, 'Le c\'est géniale j\'aimerais en faire plus vous');

INSERT posts (user_id, content) VALUES
   (2, 'Le dernier film cars est un pure banger'),
   (3, 'Je regarde un dessin anime d\'abeilles c\'est super !'),
   (1, 'Encore et encore du PHP toujours du PHP'),
   (4, 'Invaincu sur mario kart et vous ? ');
```

La syntaxe est la même, c'est la donnée user_id à laquelle il faut faire attention.
Je vous invite à exécuter la première ligne du script sql sur votre poste pour voir ce que le serveur va vous répondre :)

Créer une relation many to many

Reprenons le contexte du mini réseau social et de notre schéma de BDD conçu plutôt. Nous allons ajouter la table groups qui décrit le format qu'un groupe doit avoir et group_user représentant l'association d'utilisateurs à un ou plusieurs groupes.

Dans un premier temps, il nous faut notre table groupe.

```
DROP table IF EXISTS GROUPS;
CREATE TABLE groups (
    id INTEGER PRIMARY KEY AUTO_INCREMENT,
    NAME VARCHAR(127) NOT NULL,
    description VARCHAR(255) NOT NULL UNIQUE,
    created_at TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP,
    updated_at TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP
);
```

Dans un second temps, nous allons insérer les données dans la table «groups» .

```
INSERT INTO groups (name, description) VALUES
    ("Amoureux des oiseaux", "Ici grand fan d'oiseaux"),
    ("Les foufous du volants", "Ici on adore le vélo"),
    ("Les foufooteux", "Pro du handball"),
    ("MCF", "Mega chuper fien");
```

Enfin, nous allons créer la table qui va enregistrer l'appartenance d'un utilisateur à un groupe.

```
DROP table IF EXISTS group_user;
CREATE TABLE group_user (
    group_id INTEGER NOT NULL,
    user_id INTEGER NOT NULL,
    created_at TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP,
    CONSTRAINT fk_users_id_group_user FOREIGN KEY (user_id) REFERENCES users(id),
    CONSTRAINT fk_groups_id_group_user FOREIGN KEY (group_id) REFERENCES groups(id),
    CONSTRAINT uq_group_id_user_id_group_user UNIQUE(group_id, user_id)
);
```

La contrainte UNIQUE appliquée à plusieurs colonnes permet de garantir l'unicité d'une combinaison de valeurs dans une table.

Dans le contexte d'un système de groupes :

Elle empêche les doublons pour une même paire utilisateur/groupe
Un utilisateur peut appartenir à plusieurs groupes différents
Un groupe peut contenir plusieurs utilisateurs différents
Mais la combinaison utilisateur + groupe ne peut exister qu'une seule fois

Exemple concret :

Si l'utilisateur «Alice» (ID:1) est déjà membre du groupe «Développeurs» (ID:5), une seconde tentative d'ajout de cette même association sera rejetée par la base de données.

Insérer des données dans une table «many to many»

```
INSERT INTO group_user (group_id, user_id) VALUES
    (1, 1),
    (1, 2),
    (2, 3),
    (3, 2),
    (1, 4);
```

Exercice Pratique : Construisez et remplissez la base de données mini réseau social à l'aide des captures d'écran

Modifier des données

La commande UPDATE permet de modifier des données existantes dans une table. Voici comment l'utiliser efficacement :

```
UPDATE table_name
SET column1 = value1, column2 = value2, ...
WHERE condition;
```

```
-- Cas n°1 :  Que se passe t'il ici ?
UPDATE users SET email = 'exemple-casse-modification@exemple.org';
-- Cas n°2 : Bien
UPDATE users SET email = 'super-exemple@modification.fr' WHERE id = 1;
-- Cas n°3 : Mieux
UPDATE users SET email = 'encore-meilleur@modif-exemple.com', updated_at = CURRENT_TIMESTAMP WHERE id = 2;
```

Syntaxe de Base

- UPDATE table_name : Spécifie la table à modifier
- SET column = value : Définit les nouvelles valeurs pour les colonnes
- WHERE condition (optionnel mais crucial) : Filtre les lignes à mettre à jour

Analyse des Exemples :

Premier Cas : Sans Condition (Dangereux)

- La requête modifie tous les emails de la table users pour la même valeur.
- Impact : Tous les utilisateurs auront soudainement le même email, ce qui :
- Crée des doublons (violation de contrainte UNIQUE si elle existe)
- Rend le système inutilisable

Deuxième Cas : Avec Condition Précise

- Seul l'utilisateur avec id = 1 voit son email modifié.
- Bonnes pratiques :
  - Toujours cibler une ligne unique via son ID
  - Vérifier la condition avant exécution

Troisième Cas : Mise à Jour Multiple

- Modifie à la fois l'email et le champ updated_at pour l'utilisateur id = 2.
- Avantages :
  - Cohérence des données (horodatage automatique des modifications)
  - Opération atomique (pas besoin de deux requêtes)

Exercice Pratique : Modifier 1 information par utilisateur à l'aide de requêtes update

Supprimer des données

La commande DELETE FROM permet de supprimer définitivement des enregistrements dans une table. Sa structure minimale est :
DELETE FROM nom_table : Cible la table à modifier
WHERE condition (optionnel mais crucial) : Filtre les lignes à supprimer

```
DELETE FROM table_name WHERE condition;
```

```sql
-- Cas n°1 : Dangereux
DELETE FROM users;

-- Cas n°2 : Mieux , mais génère toujours une erreur, pourquoi ?
DELETE FROM users
WHERE email = 'mireille-danse@msn.fr';


-- Cas n°3 : Encore mieux, car on cible par identifiant mais toujours une potentielle erreur.
DELETE FROM users
WHERE id = 42;

-- Cas n°4 :
-- Explication LIKE permet d'itentifier un schéma de texte
-- conséquence on va supprimer tout les user avec un email finissant par @msn.fr
DELETE FROM users
WHERE email LIKE '%@msn.fr';


-- Cas concret de suppression
-- On supprime tout les utilisateurs qui n'ont pas modifié leurs profil depuis un an au moins
DELETE FROM users
WHERE users.updated_at < DATE_SUB(NOW(), INTERVAL 1 YEAR);
```

Cas 1 : Suppression Totale (Dangereux)

Description : Supprime tous les enregistrements de la table

Risques :

- Perte définitive de toutes les données
- Impact en cascade sur les tables liées
- Quand l'utiliser : Uniquement en développement avec des données de test

Jamais en production sans sauvegarde validée

Cas 2 : Suppression par Email (Condition Simple)

Description : Supprime via une adresse email exacte

Bon usage :

Pour des opérations de nettoyage ciblé

Cas 3 : Suppression par ID (Méthode Sécurisée)

Pourquoi c'est mieux :

- L'ID est toujours unique (1 suppression max)
- Moins sujet aux erreurs de format que l'email

Limite :

Requiert de connaître l'ID au préalable

Usage typique : Suppression depuis une interface admin ou nettoyage suite à une recherche précise

Cas 4 : Suppression par Motif Textuel (LIKE)

- Utilisation de jokers (% pour «n'importe quelle suite de caractères»)
- Exemple : %@msn.fr cible tous les emails de ce domaine

Éviter les motifs trop larges (%@% supprimerait tout)

Cas 5 : Suppression par Ancienneté (Condition Temporelle)

Supprime les enregistrements non modifiés depuis X temps

Exemple : Comtes inactifs depuis 1 an

Méthodologie Pro :

- Sauvegarde préalable obligatoire
- Utiliser une transaction pour pouvoir annuler
- Journaliser les suppressions effectuées

Problème de suppression ?
Vous avez dû remarquer que lorsqu'une table est liée avec une clé étrangère il est impossible de supprimer l'entité parent (ici users) tant que des entités enfants sont reliées (ici group_user, posts). Deux choix s'offrent à vous :

Supprimer les entités enfant avant de supprimer l'entité parents :

```sql
-- Supprime tout les posts rédigés par l'utilisateur id = 4
DELETE FROM posts WHERE user_id = 4;
-- Enlève l'utilisateur id =  4 de chaque groupe auquel il s'était inscrit
DELETE FROM group_post WHERE user_id = 4;
-- Suppression de l'utilisateur id = 4
DELETE FROM users WHERE id = 4;
```

Ou deuxième option, lors de la création des tables enfant il est nécessaire d'ajouter une option au moment de la définition de la contrainte nommée «ON DELETE CASCADE» .

```sql
CREATE TABLE posts (
   id INTEGER PRIMARY KEY AUTO_INCREMENT,
   user_id INTEGER NOT NULL,
   content VARCHAR(255) NOT NULL,
   created_at TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP,
   CONSTRAINT fk_users_id_post FOREIGN KEY (user_id) REFERENCES users(id) ON DELETE CASCADE
);

CREATE TABLE group_user (
    group_id INTEGER NOT NULL,
    user_id INTEGER NOT NULL,
    created_at TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP,
    CONSTRAINT fk_users_id_group_user FOREIGN KEY (user_id) REFERENCES users(id) ON DELETE CASCADE,
    CONSTRAINT fk_groups_id_group_user FOREIGN KEY (group_id) REFERENCES groups(id) ON DELETE CASCADE,
    CONSTRAINT uq_group_id_user_id_group_user UNIQUE(group_id, user_id)
);

-- Maintenant seul un delete suffit
DELETE FROM users WHERE id = 4;
```

Note : Vous pouvez également modifier la structure d'une table avec la commande «ALTER TABLE», on verra comment faire ultérieurement.

Exercice Pratique : Supprimer 1 utilisateur à l'aide de requêtes delete

Select

Jusqu'à présent nous avons vu comment créer/modifier/supprimer. Maintenant, il est temps de voir comment récupérer nos données pour par la suite les utiliser dans nos projets web.

La commande SELECT permet de récupérer une ou plusieurs colonnes d'une ou plusieurs tables.
Commençons par la requête SQL la plus simple :

```sql
SELECT * FROM users;
```

Cette requête permet d'extraire toutes les données de la table users.

SELECT * :

L'astérisque (*) est un raccourci pour demander toutes les colonnes (identifiant, nom, email, etc.) sans avoir à les lister une par une.

FROM users :

Spécifie la table source des données, ici users. Cette précision est indispensable pour que le système sache où chercher les informations.

id	username	email	password	created_at
1	tibone	tibtib76@hotmail.fr	$2y$10$KEWDQUGrhOJlrp8opunTz.VwrsUB2Rn9FWLKdU...	2025-03-28 13:20:47
2	alex-l	alexis.l.msg@gmail.com	$2y$10$941AoVHhR/wGZdI5aSdb6.WjJdKrC9g47fuwjXZ...	2025-03-28 13:20:47
3	miguel	migoumac@yahoo.com	$2y$10$x.Sysj/rJCzFJyg9CKZTEOftlGDaXwkTiYly.0o30tS...	2025-03-28 13:20:47
4	maya-labeille	mireille-danse@msn.fr	$2y$10$wUCQTkZzmWjW9isQVVB.E.GpU.C4c1vA2QFhT...	2025-03-28 13:20:47
5	louis-sanson	hein-quoi@gmail.com	$2y$10$wUCQTkZzmWjW9isQVVB.E.GpU.C4c1vA2QFhT...	2025-03-28 13:20:47

C'est un bon début, mais on n'a pas tout le temps besoin de charger la totalité de nos colonnes, parfois nous avons besoin uniquement d'afficher certaines informations. Reprenons notre table user, pour notre projet mini réseau social. Cette fois-ci nous souhaitons récupérer et afficher le username et l'email sur la page de profil. Nous allons donc construire une requête qui va chercher uniquement ces informations.

```
SELECT username, email FROM users;
```

Il suffit tout simplement de SELECT les colonnes désirées avec une virgule pour les séparer.

username	email
tibone	tibtib76@hotmail.fr
alex-l	alexis.l.msg@gmail.com
miguel	migoumac@yahoo.com
maya-labeille	mireille-danse@msn.fr
louis-sanson	hein-quoi@gmail.com

Where

Maintenant que vous maîtrisez la sélection des colonnes, passons au filtrage des données pour éviter de charger inutilement toute une table. La clause WHERE (qui signifie «où» en français) permet de cibler uniquement les enregistrements répondant à des critères spécifiques.
Prenons un exemple concret dans votre mini-réseau social : vous souhaitez afficher la date d'inscription de l'utilisateur «miguel». Sans filtre, une requête SELECT * FROM users renverrait tous les utilisateurs, ce qui serait inefficace. Avec WHERE, vous pouvez préciser : «Récupère uniquement les données de l'utilisateur dont le pseudo est 'miguel'».

```
SELECT id, username, created_at FROM users WHERE username = 'miguel';
```

#	id	username	created_at
1	3	miguel	2025-03-28 13:20:47

Dans l'idéal on préfère ajouter des WHERE sur l'id car c'est un identifiant unique.

```
SELECT id, username, created_at FROM users WHERE id = 3;
```

Opérateurs de comparaison

Si l'opérateur = permet de filtrer les valeurs exactes (comme trouver l'utilisateur «miguel»), SQL offre d'autres outils pour des requêtes plus puissantes et nuancées. Voici les principaux opérateurs à connaître :

1. Opérateurs de Base

- = : Égalité (ex : username = 'miguel')
- != ou <> : Différence (ex : status != 'banned')
- > / < : Supérieur / Inférieur (ex : age > 18)
- >= / <= : Supérieur ou égal / Inférieur ou égal (ex : created_at <= '2023-01-01')

2. Pour les Textes et Recherches Approchées

- LIKE : Recherche par motif (ex : email LIKE '%@gmail.com')
- % = n'importe quelle suite de caractères

_ = un seul caractère inconnu
IN : Valeurs dans une liste (ex : id IN (1, 5, 12))

3. Pour les Dates et Plages de Valeurs

BETWEEN : Plage inclusive (ex : age BETWEEN 20 AND 30)
IS NULL : Vérifie l'absence de valeur (ex : updated_at IS NULL)

Note : La base de données mini_reseau_social a été enrichie pour permettre des requêtes plus pertinentes et mieux refléter un environnement réel. Les améliorations suivantes ont été implémentées :
Table users :
Ajout de la colonne verified_at (TIMESTAMP) : enregistre la date de vérification de l'adresse email de l'utilisateur
Ajout de la colonne is_banned (BOOL) : indique si le compte est suspendu ou non
Ajout de la colonne age (TINYINT) : stocke l'âge de l'utilisateur
Un jeu de données a également été inséré pour plus de données.

Exemples pratique :

```sql
-- On récupère :
--   - id
--   - le username
--   - le mail
--   - la date de création du compte
--   - l'état de son banissement
-- des utilisateurs bannis
SELECT id, username, email, created at, is banned FROM users WHERE is banned != FALSE;
```

Cette requête permet de lister les utilisateurs bannis du réseau social. Plus précisément, elle récupère pour chaque compte concerné : son identifiant unique, son pseudonyme, son adresse email, sa date d'inscription et son statut de bannissement.

La partie clé se trouve dans la condition WHERE is_banned != FALSE. Ici, l'opérateur != signifie «différent de». La requête filtre donc les utilisateurs dont le statut de bannissement n'est pas égal à «FALSE», c'est-à-dire ceux qui sont bannis (is_banned = TRUE). En d'autres termes, elle exclut automatiquement tous les comptes actifs (non bannis) pour ne garder que ceux sanctionnés.

```sql
-- On récupère :
--   - id
--   - le username
--   - l'âge
--   - la date de création du compte
-- des utilisateurs âgés d'au moins 21 ans.
SELECT id, username, age, created_at FROM users WHERE age > 20;
```

Cette requête permet de lister les utilisateurs majeurs de 21 ans et plus sur le réseau social. Elle affiche pour chaque profil concerné : son numéro d'identifiant, son pseudonyme, son âge exact et sa date d'inscription.

La condition WHERE age > 20 est cruciale ici. L'opérateur > signifie «strictement supérieur à». La requête filtre donc tous les utilisateurs dont l'âge est supérieur à 20 ans, ce qui équivaut à sélectionner ceux qui ont 21 ans ou plus.

```sql
-- On récupère :
-- - id
-- - le username
-- - l'âge
-- - la date de création du compte
SELECT id, username, age, created_at FROM users WHERE age < 18;
```

Cette requête permet de lister spécifiquement les utilisateurs mineurs (moins de 18 ans) inscrits sur le réseau social. Pour chacun d'eux, elle affiche quatre informations principales : leur numéro d'identifiant unique, leur pseudonyme, leur âge exact et la date à laquelle ils ont créé leur compte.

La partie importante est la condition WHERE age < 18. L'opérateur < signifie «strictement inférieur à». Cela permet de filtrer et de ne sélectionner que les utilisateurs dont l'âge est inférieur à 18 ans, excluant ainsi automatiquement tous les membres majeurs.

```sql
-- On récupère :
-- - id
-- - Le username
-- - L'email
-- - La date de création du compte
-- des utilisateurs dont l'email commence par un a
SELECT id, username, email, created_at FROM users WHERE email LIKE 'a%';
```

Cette requête permet de rechercher spécifiquement les utilisateurs dont l'adresse email commence par la lettre «a». Pour chaque compte correspondant, elle affiche quatre informations principales : l'identifiant unique, le pseudonyme, l'adresse email complète et la date de création du compte.

La partie essentielle est la condition WHERE email LIKE 'a%'. Ici, l'opérateur LIKE combiné au motif 'a%' fonctionne comme un filtre intelligent :

- Le symbole % est un caractère «joker» qui signifie «n'importe quelle suite de caractères»
- La combinaison 'a%' se lit donc : «commençant par 'a' suivi de n'importe quoi»

Opérateurs logique

En SQL, les opérateurs logiques permettent de combiner ou modifier des conditions dans vos requêtes pour créer des filtres plus complexes.

Il en existe 3 en SQL : AND, OR, NOT.

AND (ou ET)

Cet opérateur logique combine plusieurs critères.
Par exemple si nous souhaitons récupérer dans notre base de données «les utilisateurs majeurs et bannis ».

```sql
-- On récupère tout
-- Des utilisateurs bannis et âgés d'au moins 18 ans
SELECT * FROM users WHERE is_banned = FALSE AND age > 17;
```

Pour qu'un utilisateur soit affiché dans la requête, il faut nécessairement qu'il respecte les deux conditions. C'est le rôle de l'opérateur AND (et).

Grâce à cet opérateur vous pouvez créer des requêtes avec autant de AND opérateur que nécessaire pour faire un filtre plus efficace/détaillé.

```sql
SELECT * FROM users WHERE age > 17 AND is_banned = FALSE AND created_at < DATE_SUB(NOW(), INTERVAL 3 MONTH);
```

Ici on utilise 2 AND opérateur pour créer un filtre plus strict, l'utilisateur affichable doit répondre à 3 critères :

- Majeur
- Non banni
- Avoir un compte avec au moins 3 mois d'ancienneté

Ce genre de requête est utilisé dans le monde professionnel, par exemple pour désigner les utilisateurs éligibles à un concours d'une plateforme.

OR (OU)

L'opérateur OR, comme l'opérateur AND à pour but de combiner plusieurs conditions. Contrairement à AND, OR est plus flexible, il suffit qu'une seule condition soit vraie pour inclure le résultat.

```sql
-- On récupère tout
-- Des posts dont le contenu possède la chaîne de caractères jeu ou dont le contenu possède la chaîne de caractères mario.
SELECT * FROM posts WHERE content LIKE '%jeu%' OR content LIKE '%mario%';
```

On peut traduire cette requête par « affiche moi tous les posts dont le contenu contient la chaîne de caractères « jeu » **OU** dont le contenu contient la chaîne de caractères « mario ».

L'opérateur vérifie que l'une des deux expressions adjacentes soit vraie. Elle permet contrairement à ET, d'élargir votre recherche.

NOT (non en français)

L'opérateur NOT **inverse une condition**, cela permet d'exclure les résultats qui correspondent au critère spécifié. Donc si une condition est vraie, NOT la rend fausse et inversement.

```sql
-- On récupère tout
-- Des utilisateurs dont le username contient un a
SELECT * FROM users WHERE username LIKE '%a%';

-- On récupère tout
-- Des utilisateurs dont le username ne contient pas un a
SELECT * FROM users WHERE NOT username LIKE '%a%';
```

NOT sert donc à faire de l'exclusion ciblée.

De plus, tous les éléments que nous venons de voir, (ET, OR, NOT) peuvent se combiner pour créer des filtres complexes.

```sql
SELECT * FROM users WHERE
   NOT age < 18
   AND created_at <= DATE_SUB(NOW(), INTERVAL 48 DAY)
   OR username = 'admin';
```

Cette requête cible les utilisateurs majeurs, avec un compte d'au moins 48 jours d'ancienneté, ou si le username est admin.

Les fonctions d'aggrégats

Les fonctions d'agrégation sont des outils puissants en SQL permettant d'effectuer des calculs sur un ensemble de lignes pour en tirer des résultats synthétiques. Elles servent à analyser des données en les résumant via des opérations statistiques ou mathématiques, plutôt que de les afficher une par une.

Voici une liste des plus utilisées :

- Compter des éléments (COUNT)
- Calculer des sommes (SUM), des moyennes (AVG)
- Trouver des valeurs extrêmes (MIN, MAX)
- Grouper des données pour une analyse segmentée (GROUP BY)

Exemples pratique :

```sql
SELECT COUNT(*) FROM users;
SELECT COUNT(*) FROM users WHERE is_banned = FALSE;
```

#	COUNT(*)
1	20

#	COUNT(*)
1	16

La fonction COUNT() est la plus simple et la plus utilisée des fonctions d'agrégation en SQL. Elle permet de compter le nombre de lignes correspondant à une condition donnée.

```sql
-- SUM permet de faire la somme de l'âge de tous les utilisateurs de la BDD
SELECT SUM(age) FROM users;

-- Si on utilise la fonction COUNT vu précédemment on peut calculer l'âge moyen de nos utilisateurs
SELECT SUM(age)/COUNT(id) FROM users;
```

#	SUM(age)
1	674

#	SUM(age)/COUNT(id)
1	33,7

La fonction SUM() (somme en français) est une fonction d'agrégation essentielle en SQL, utilisée pour additionner les valeurs numériques d'une colonne.

À quoi sert SUM() ?
- Calculer un total cumulé (ex : chiffre d'affaires, likes, temps de connexion).
- Travailler sur des données numériques (INT, DECIMAL, etc.).

```sql
SELECT AVG(age) FROM users;

SELECT AVG(age) FROM users WHERE is_banned = TRUE;
```

#	AVG(age)
1	33,7

#	AVG(age)
1	25,25

La fonction AVG() (moyenne en français) est une fonction d'agrégation fondamentale qui calcule la moyenne arithmétique des valeurs numériques d'une colonne.

À quoi sert AVG() ?
- Calculer une valeur centrale représentative (âge moyen, note moyenne, etc.)
- Analyser des tendances sur des données numériques

Note : On utilise la fonction average pour déterminer l'âge moyen de nos utilisateurs dans deux cas : l'âge moyen général de nos utilisateurs et l'âge moyen des utilisateurs bannis.

```sql
SELECT MIN(age), MAX(age) FROM users;
```

#	MIN(age)	MAX(age)
1	14	112

Les fonctions MIN() et MAX() sont des fonctions d'agrégation qui permettent respectivement de trouver les valeurs minimales et maximales dans une colonne.

À quoi servent MIN() et MAX() ?
- Identifier les bornes extrêmes d'un jeu de données
- Trouver des records (plus ancien/plus récent, plus petit/plus grand)
- Analyser l'étendue des valeurs (MAX() - MIN() donne la plage totale)

```sql
SELECT user_id, COUNT(*) FROM posts GROUP BY user_id;
```

#	user_id	COUNT(*)
1	1	11
2	2	11
3	3	11
4	4	11
5	5	10
6	6	10
7	7	10
8	8	10
9	9	10
10	10	10
11	11	10
12	12	10
13	13	10
14	14	10
15	15	10

La clause GROUP BY est un outil fondamental en SQL qui permet de grouper des lignes partageant une même caractéristique pour effectuer des calculs agrégés sur chaque groupe.

À quoi sert GROUP BY ?
- Regrouper des données identiques ou similaires
- Combiner avec des fonctions d'agrégation (COUNT, SUM, AVG, etc.)
- Analyser des données par catégories ou segments
- Réduire plusieurs lignes en résultats synthétiques

Note : La requête nous permet d'avoir un résultat soigné pour afficher le nombre de posts faits par utilisateurs.

```
SELECT user_id, COUNT(*) FROM group_user GROUP BY user_id;
```

#	user_id	COUNT(*)
1	1	4
2	2	4
3	3	4
4	4	4
5	5	3
6	6	4
7	7	4
8	8	3
9	9	4
10	10	4
11	11	3
12	12	2
13	13	3
14	14	3
15	15	3

Ici on se sert du GROUP BY pour afficher le nombre de groupes rejoints par chaque utilisateur.

Les jointures

Les jointures (ou JOIN en anglais) sont l'un des concepts les plus puissants en SQL. Elles permettent de relier des données provenant de plusieurs tables entre elles, comme si vous assembliez les pièces d'un puzzle !

Reprenons notre projet mini réseau social :
- On a une table users
- On a une table posts

Sans jointure, vous ne sauriez pas qui a posté quel statut. Avec une jointure, vous pouvez facilement associer ces informations en reliant les tables grâce à un champ commun (comme un user_id).

Pourquoi c'est utile ?
- Éviter la répétition des données (principe des bases de données relationnelles)
- Combiner des informations dispersées dans différentes tables
- Répondre à des questions complexes du type «Quels utilisateurs ont rejoint tel groupe?»

```
SELECT users.username, posts.created_at, posts.content
FROM users
JOIN posts ON posts.user_id = users.id;
```

Que fait cette requête ?

1. Elle relie deux tables :
- users (contient les informations des membres)
- posts (contient les publications)
2. Mécanisme de jointure :
- La clause JOIN posts ON posts.user_id = users.id fonctionne comme une agrapheuse :
- Elle associe chaque post (posts) à son auteur (users)
- En comparant les clés : user_id (dans posts) = id (dans users)
3. Résultat :
- Le résultat affichera pour chaque publication :
  - Le pseudonyme de l'auteur (users.username)
  - La date de création du post (posts.created_at)
  - Le contenu du message (posts.content)

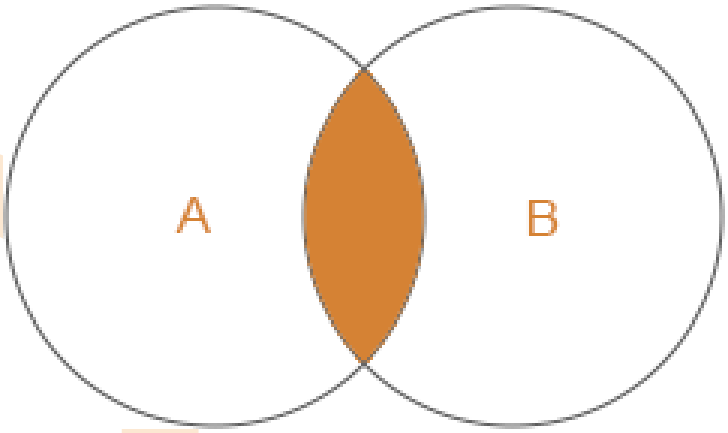
Les types de JOIN

Type de JOIN	Comportement	Usage typique
INNER JOIN	Uniquement les correspondances	Donne-moi les users AVEC les posts
LEFT JOIN	Tous à gauche + correspondance à droite	Donne-mois tous les users, même s'ils n'ont pas de commande
RIGHT JOIN	Tous à droite + correspondance à gauche	(Rare en pratique)

Note : Il existe encore d'autres types de JOIN mais ne sont pas tous supportés par MySQL, vous pourrez trouver la liste ici.

INNER JOIN

Le INNER JOIN (ou simplement JOIN) est le type de jointure le plus courant. Il ne retourne que les lignes où il y a une correspondance dans les deux tables.




```
SELECT colonnes
FROM tableA
INNER JOIN tableB ON tableA.clef = tableB.clef
```

Mécanisme :

- SQL examine chaque ligne de table A
- Pour chaque ligne, il cherche les correspondances dans la table B via la condition ON
- Seules les paires de lignes qui matchent sont incluses dans le résultat

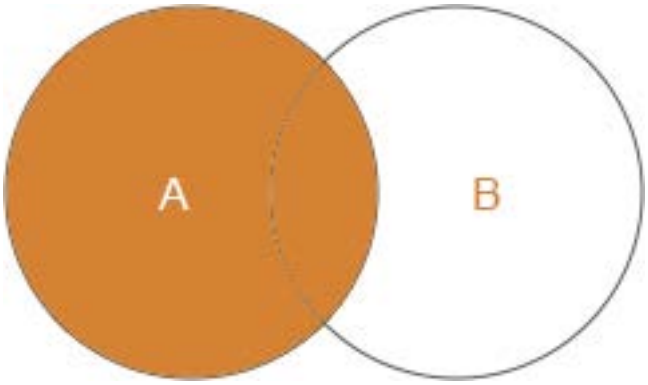
Exemple :

```
SELECT users.username, posts.created_at, posts.content
FROM users
JOIN posts ON posts.user_id = users.id;
```

Seuls les users qui auront des posts associés seront affichés.

LEFT JOIN

Le LEFT JOIN (ou LEFT OUTER JOIN) retourne toutes les lignes de la table de gauche, même si aucune correspondance n'existe dans la table de droite. Les colonnes de la table de droite seront remplies avec NULL en cas d'absence de correspondance.



```sql
SELECT colonnes
FROM tableA
LEFT JOIN tableB ON tableA.clef = tableB.clef
```

Mécanisme :

- SQL examine chaque ligne de table A
- Pour chaque ligne, il cherche les correspondances dans la table B via la condition ON
- Toutes les lignes de la table A seront affichées
- Seules les lignes de la table B qui matchent sont incluses dans le résultat

```sql
SELECT users.username, posts.created_at, posts.content
 FROM users
 LEFT JOIN posts ON posts.user_id = users.id;
```

Grâce à cette requête, on peut cibler aussi les utilisateurs qui n'ont jamais écrit de post en l'améliorant un peu.

```sql
SELECT users.username, posts.created_at, posts.content
 FROM users
 LEFT JOIN posts ON posts.user_id = users.id
 WHERE posts.content IS NULL;
```

#	username	created_at	content
1	claire_m	(NULL)	(NULL)
2	jessica_p	(NULL)	(NULL)
3	lucie_b	(NULL)	(NULL)
4	quentin_g	(NULL)	(NULL)
5	romain_f	(NULL)	(NULL)

Les jointures (many to many)

Quand un utilisateur peut appartenir à plusieurs groupes et qu'un groupe peut contenir plusieurs utilisateurs, on a besoin de 3 tables :

- users Liste des membres
- groups Liste des groupes
- group_user Table de liaison (qui enregistre qui est dans quel groupe

Question : Comment lier ces 3 tables ?

Solution : Avec plusieurs JOIN

```sql
SELECT users.username, groups.name FROM users
INNER JOIN group_user ON users.id = group_user.user_id
INNER JOIN groups ON groups.id = group_user.group_id;
```

Toujours 2 INNER JOIN pour traverser une relation many-to-many :

- Premier JOIN : table principale table de liaison
- Second JOIN : table de liaison table associée

Pour vous aider à comprendre comment faire des join à travers plusieurs tables, il faut s'imaginer que chaque JOIN est un pont d'une table à une autre, et que pour créer ce pont il faut avoir une clé (étrangère) en commun.

Exercice Pratique : Allez voir le fichier **/2_SQL/exercices_pratique/consignes/consignes_ep_1.md**