Commandes SQL

CREATE

Permet de créer une base de donnée ou une table

Base de données

CREATE DATABASE DBMABASE;

USE DBMABASE; permet d'accéder à la base afin d'y travailler

Table

```
CREATE TABLE nom_de_la_table
(
        colonne_1   type_donnees,
        colonne_2   type_donnees,
        colonne_3   type_donnees,
        ...
        colonne_n   type_donnees
)
```

Pour chaque colonne, il est également possible de définir des options telles que :
• **NOT NULL :** empêche d'enregistrer une valeur nulle pour une colonne.
• **DEFAULT :** attribuer une valeur par défaut si aucune données n'est indiquée pour cette colonne lors de l'ajout d'une ligne dans la table.
• **PRIMARY KEY :** indiquer si cette colonne est considérée comme clé primaire pour un index.

- **création d'une table avec clé primaire**

```
CREATE TABLE employe
(
        id_utilisateur INT(6) AUTO_INCREMENT PRIMARY KEY NOT NULL,
        nom VARCHAR(50),
        prenom VARCHAR(50),
        email VARCHAR(70),
        date_naissance DATE
)
```

```
CREATE TABLE MONUMENT (
        ID_MONUMENT INT  auto_increment,
        ID_TYPESITE CHAR (5) NOT NULL,
        NOM VARCHAR (100) NOT NULL,
        ADRESSE VARCHAR (100) NOT NULL,
        creer_le TIMESTAMP DEFAULT CURRENT_TIMESTAMP
        primary key (ID_MONUMENT)
)
```

- **création d'une table avec clé étrangère**

```
CREATE TABLE CLIENT (
        Id_Client int NOT NULL,
        …,
        PRIMARY KEY (Id_Client)
)

CREATE TABLE PRODUIT(
        Id_Produit NOT NULL,
        …,
        Id_Client_fk int,
        PRIMARY KEY (Id_Produit)
        FOREIGN KEY (Id_Client_fk) REFERENCES CLIENT(Id_Client)
);
```

DROP

Permet de supprimer une base de donnée ou une table

```
DROP DATABASE DBMONUMENT

DROP TABLE PRODUIT
```

INSERT INTO

Permet d'insérer des données dans la table. Cette commande permet au choix d'inclure une seule ligne a la base existante ou plusieurs lignes d'un coup.

INSERT INTO table
VALUES ('valeur 1', 'valeur 2', …)

INSERT INTO table
(nom_colonne_1, nom_colonne_2, ...
VALUES ('valeur 1', 'valeur 2', …)

CONSTRAINTS

Les contraintes SQL sont utilisées pour spécifier des règles pour les données d'une table.

Les contraintes sont utilisées pour limiter le type de données pouvant entrer dans une table. Cela garantit l'exactitude et la fiabilité des données du tableau. S'il y a une violation entre la contrainte et l'action de données, l'action est abandonnée.

Les contraintes peuvent être au niveau de la colonne ou au niveau de la table. Les contraintes de niveau de colonne s'appliquent à une colonne et les contraintes de niveau de table s'appliquent à l'ensemble du tableau.

Les contraintes suivantes sont couramment utilisées dans SQL:

NOT NULL - Garantit qu'une colonne ne peut pas avoir une valeur NULL
UNIQUE - Garantit que toutes les valeurs d'une colonne sont différentes
PRIMARY KEY - Identifie de manière unique chaque ligne d'une table
FOREIGN KEY - Identifie de manière unique une ligne / un enregistrement dans une autre table
CHECK - Garantit que toutes les valeurs d'une colonne satisfont à une condition spécifique
DEFAULT - Définit une valeur par défaut pour une colonne lorsqu'aucune valeur n'est spécifiée
INDEX - Utilisé pour créer et récupérer des données de la base de données très rapidement

ALTER TABLE

Permet de modifier une table existante. Il est ainsi possible d’ajouter une colonne, d’en supprimer une ou de modifier une colonne existante, par exemple pour changer le type.

- **ajouter une colonne**

ALTER TABLE utilisateur
ADD adresse_rue VARCHAR(50)

- **supprimer une colonne**

ALTER TABLE utilisateur
DROP COLUMN adresse_rue

- **modifier une colonne**

ALTER TABLE utilisateur
MODIFY COLUMN adresse_rue varchar(100)

- **renommer une colonne**

ALTER TABLE utilisateur
CHANGE adresse_rue adresse varchar(100)

- **Contraintes avec alter table**

ALTER TABLE employe ADD PRIMARY KEY (id_employe)

ALTER TABLE employe
ADD CONSTRAINT pk_employe PRIMARY KEY (id_employe)

ALTER TABLE employe
ADD CONSTRAINT pk_employe PRIMARY KEY (id_employe,nom)

ALTER TABLE facture
ADD FOREIGN KEY (id_client_fk) REFERENCES client(id_client)

ALTER TABLE facture
ADD CONSTRAINT fk_clientfacture
FOREIGN KEY (id_client_fk) REFERENCES client(id_client)

Pour supprimer une contrainte on utilise la commande DROP

ALTER TABLE employe DROP PRIMARY KEY

```
ALTER TABLE employe
        DROP CONSTRAINT pk_employe

ALTER TABLE facture
DROP FOREIGN KEY

ALTER TABLE facture
 DROP FOREIGN KEY fk_clientfacture
```

UPDATE

Permet d'effectuer des modifications sur des lignes existantes

```
UPDATE table
SET colonne_1 = 'nouvelle valeur'
WHERE condition

UPDATE table
SET colonne_1 = 'valeur 1', colonne_2 = 'valeur 2', colonne_3 = 'valeur 3'
WHERE condition
```

DELETE

Permet d'effectuer des modifications sur des lignes existantes

```
DELETE FROM table
WHERE condition
```

Les 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()** pour calculer la moyenne sur un ensemble d'enregistrement
- **COUNT()** pour compter le nombre d'enregistrement sur une table ou une colonne distincte
- **MAX()** pour récupérer la valeur maximum d'une colonne sur un ensemble de ligne. Cela s'applique à la fois pour des données numériques ou alphanumérique
- **MIN()** pour récupérer la valeur minimum de la même manière que MAX()
- **SUM()** pour calculer la somme sur un ensemble d'enregistrement

Les sous requêtes

Dans le langage SQL une sous requête (aussi appelé “requête imbriquée” ou “requête en cascade”) consiste à exécuter une requête à l’intérieur d’une autre requête. Une requête imbriquée est souvent utilisée au sein d’une clause WHERE ou de HAVING pou remplacer une ou plusieurs constante.

Une requête imbriquée peut renvoyer trois types de résultats :

- une valeur scalaire
- une colonne
- une table

SELECT

Permet de sélectionner les données dans une table.

SELECT * | [ALL] | [DISTINCT]
FROM table
WHERE condition
***GROUP BY ex*pression**
HAVING condition
{ UNION | INTERSECT | EXCEPT }
ORDER BY expression
LIMIT count
OFFSET start

WHERE

La commande WHERE dans une requête SQL permet d’extraire les lignes d’une base de données qui respectent une condition. Cela permet d’obtenir uniquement les informations désirées.

Pour poser les conditions on utilise les opérateurs de comparaison. La liste suivante présente quelques uns des opérateurs les plus couramment utilisés.

=	***Égale***
<>	***Pas égale***
!=	***Pas égale***
>	***Supérieur à***
<	***Inférieur à***
>=	***Supérieur ou égale à***

<=	*Inférieur ou égale à*
IN	*Liste de plusieurs valeurs possibles*
BETWEEN	*Valeur comprise dans un intervalle donnée (utile pour les nombres ou dates)*
LIKE	*Recherche en spécifiant le début, milieu ou fin d'un mot.*
IS NULL	*Valeur est nulle*
IS NOT NULL	*Valeur n'est pas nulle*

AND et OR

Les opérateurs logiques AND et OR peuvent être utilisées au sein de la commande WHERE pour combiner des conditions.

```
SELECT nom_colonnes
FROM nom_table
WHERE condition1 AND condition2
```

DISTINCT

Permet d'éviter des redondances dans les résultats

```
SELECT DISTINCT ma_colonne
FROM nom_du_tableau
```

AS

Permet de renommer temporairement une colonne ou une table dans une requete. Cette astuce est particulièrement utile pour faciliter la lecture des requêtes.

```
SELECT colonne1 AS c1, colonne2
FROM table
```

```
SELECT colonne1 c1, colonne2
FROM table
```

```
SELECT *
FROM nom_table AS t1
```

```
SELECT *
FROM table t1
```

IN

L'opérateur logique IN dans SQL s'utilise avec la commande WHERE pour vérifier si une colonne est égale à une des valeurs comprise dans une liste de valeurs déterminées.

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

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.

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

LIKE

Permet d'effectuer une recherche sur un modèle particulier.

SELECT *
FROM table
WHERE colonne LIKE modele

Les modèles peuvent généralement être définis de la manière suivante :
• **LIKE '%a'** : le caractère ≪ % ≫ est un caractère joker qui remplace tous les autres caractères. Ainsi, ce modèle permet de rechercher toutes les chaînes de caractère qui se termine par un ≪a ≫.
• **LIKE 'a%'** : ce modèle permet de rechercher toutes les lignes de ≪ colonne ≫ qui commence par un ≪ a ≫.
• **LIKE '%a%'** : ce modèle est utilisé pour rechercher tous les enregistrements dont la colonne contient ≪ a ≫.
• **LIKE 'pa%on'** : ce modèle permet de rechercher les chaînes qui commence par ≪ pa ≫ et qui se terminent par ≪ on ≫, comme ≪ pantalon ≫ ou ≪ pardon ≫.
• **LIKE 'a_c'** : peu utilise, le caractère ≪ _ ≫ (underscore) peut être remplacé par n'importe quel caractère, mais un seul caractère uniquement

GROUP BY

La commande GROUP BY est utilisée en SQL pour grouper plusieurs résultats et utiliser une fonction scalaire sur un groupe de résultat.

SELECT colonne1, fonction(colonne2)
FROM table
GROUP BY colonne1

HAVING

La condition HAVING en SQL est presque similaire à WHERE à la seule différence que HAVING permet de filtrer en utilisant des fonctions scalaires.

SELECT colonne1, fonction(colonne2)
FROM nom_table
GROUP BY colonne1
HAVING fonction(colonne2) opérateur valeur

HAVING est très souvent utilisé en même temps que GROUP BY bien que ce ne soit pas obligatoire.

ORDER BY

La commande ORDER BY permet de trier les lignes issues d'une requête SQL. Il est possible de trier les données sur une ou plusieurs colonnes, par ordre ascendant ou descendant.

SELECT colonne1, colonne2
FROM table
ORDER BY colonne1

Par défaut les résultats sont classés par ordre ascendant, toutefois il est possible d'inverser l'ordre en utilisant le suffixe DESC après le nom de la colonne. Par ailleurs, il est possible de trier sur plusieurs colonnes en les séparant par une virgule.

SELECT colonne1, colonne2, colonne3
FROM table
ORDER BY colonne1 DESC, colonne2 ASC

ANY

Permet de comparer une valeur avec le résultat d'une sous requête.

```
SELECT *
FROM table1
WHERE condition opérateur ANY (
        SELECT *
        FROM table2
        WHERE condition2
)
```

La commande IN est équivalent à l'opérateur = suivi de ANY.

LIMIT / OFFSET

Permet de spécifier le nombre maximum de résultats que l'ont souhaite obtenir. Cette clause est souvent associe à un OFFSET, c'est-a-dire effectuer un décalage sur le jeu de résultat. Ces 2 clauses permettent par exemple d'effectuer des système de pagination.

```
SELECT *
FROM table
LIMIT nombre
OFFSET nombre
```

Jointure SQL

Les jointures en SQL permettent d'associer plusieurs tables dans une même requête. Cela permet d'exploiter la puissance des bases de données relationnelles pour obtenir des résultats qui combinent les données de plusieurs tables de manière efficace.

En général, les jointures consistent a associer des lignes de 2 tables en associant l'égalité des valeurs d'une colonne d'une première table par rapport à la valeur d'une colonne d'une seconde table.

Types de jointures

Il y a plusieurs méthodes pour associer 2 tables ensemble. Voici la liste des différentes techniques qui sont utilisées :
• **INNER JOIN** : jointure interne pour retourner les enregistrements quand la condition est vrai dans les 2 tables. C'est l'une des jointures les plus communes.

• **CROSS JOIN** : jointure croisée permettant de faire le produit cartésien de 2 tables. En d'autres mots, permet de joindre chaque lignes d'une table avec chaque lignes d'une seconde table. Attention, le nombre de résultats est en général très élevé.
• **LEFT JOIN** (ou **LEFT OUTER JOIN**) : jointure externe pour retourner tous les enregistrements de la table de gauche (LEFT = gauche) même si la condition n'est pas vérifiée dans l'autre table.
• **RIGHT JOIN** (ou **RIGHT OUTER JOIN**) : jointure externe pour retourner tous les enregistrements de la table de droite (RIGHT = droite) même si la condition n'est pas vérifiée dans l'autre table.
• **FULL JOIN** (ou **FULL OUTER JOIN**) : jointure externe pour retourner les résultats quand la condition est vrai dans au moins une des 2 tables.
• **SELF JOIN** : permet d'effectuer une jointure d'une table avec elle-même comme si c'était une autre table.
• **NATURAL JOIN** : jointure naturelle entre 2 tables s'il y a au moins une colonne qui porte le même nom entre les 2 tables SQL
• **UNION JOIN** : jointure d'union

Les jointures les plus souvent utilisées sont :

➔
SELECT *
FROM table1
INNER JOIN table2 ON table1.id = table2.fk_id

Cette requête permet de sélectionner les enregistrements des tables table1 et table2 lorsque les données de la colonne id de table1 est égal aux données de la colonne fk_id de table2.

INNER JOIN est identique à WHERE

➔
SELECT *
FROM table1
LEFT JOIN table2 ON table1.id = table2.fk_id

Cette requête permet de lister les enregistrement de table1, même s'il n'y a pas de correspondance avec table2. Les lignes où il n'y a pas de correspondance les colonnes de table2 vaudront toutes NULL.

➔
SELECT *
FROM table1
RIGHT JOIN table2 ON table1.id = table2.fk_id

Cette syntaxe stipule qu'il faut lister toutes les lignes du tableau table2 (tableau de droite) et afficher les données associées du tableau table1 s'il y a une correspondance entre ID de table1 et FK_ID de table2. S'il n'y a pas de correspondance, l'enregistrement de table2 sera affiche et les colonnes de table1 vaudront toutes NULL.

CREATE INDEX

Permet de créer un index. L'index est utile pour accélérer l'exécution d'une requête SQL.

CREATE INDEX index_nom ON table (colonne1,colonne2,...)

Un index unique permet de spécifier qu'une ou plusieurs colonnes doivent contenir des valeurs uniques à chaque enregistrement.

CREATE UNIQUE INDEX index_nom ON table (`colonne1,colonne2,...);