

I. Terminologie et les concepts fondamentaux

Une **base de données relationnelle (BDR)** est un système de gestion de données qui organise les informations sous forme de **tables**, où chaque table est constituée de lignes et de colonnes. Les lignes représentent des **enregistrements** (ou tuples) et les colonnes des **attributs** (ou champs). Le modèle relationnel repose sur des **relations** définies entre ces tables, souvent par l'intermédiaire de **clés primaires** et de **clés étrangères**.

Exemple : Voici les données d'une boutique de ventes des produits IT

Table client

idClient	Nom	Email
201	Ahmed Nebli	Nebli.ah@yahoo.fr
202	Layla Gafsi	Gafsi.m@gmail.com
203	Fatima Ayari	ayari@topnet.net
204	Omar Ben Ammar	bjAmmar@gmail.com
205	Salma Nasri	Nasri.salma@outlook.com

Table Article

ArticleID	Nom Article	Marque	Prix en dinar Tunisien
101	Ordinateur portable	HP	800
102	Smartphone	Huawei	500
103	Tablette	Infinix	300
104	Écouteurs Bluetooth	GBL	50
105	Clavier mécanique	Logitech	100

Table achat

idClient	ArticleID	Date d'Achat	Quantité
202	101	2024-01-15	1
201	102	2024-01-16	1
203	103	2024-02-10	2
202	104	2024-03-05	3
205	105	2024-04-20	1

On peut dégager la représentation textuelle suivante :

client(idClient, nom, email)

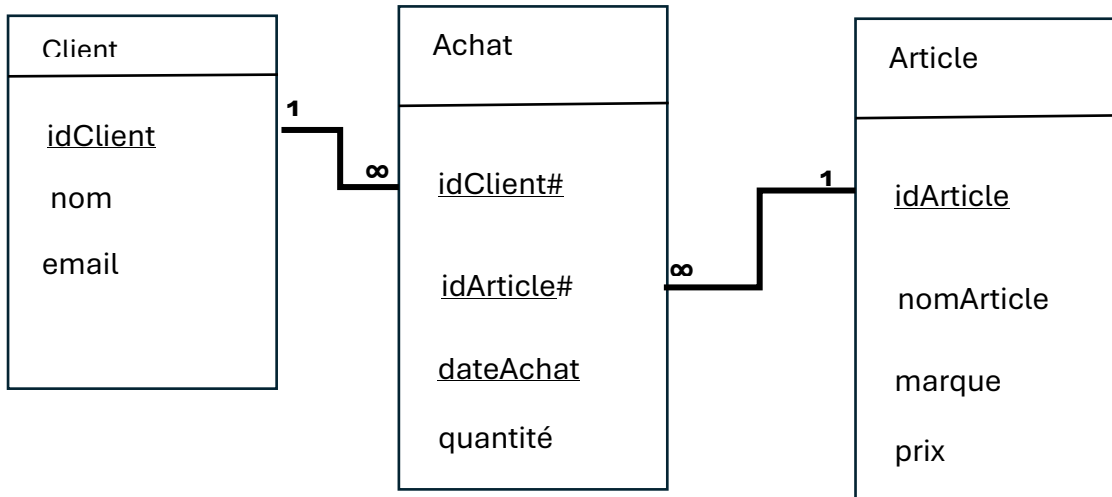
article(idArticle, nomArticle, marque, prix)

achat (idClient#, idArticle#, dateAchat, quantité)

Remarques :

- ❖ Les clés primaires sont soulignées
- ❖ Les clés étrangères sont identifiées par "#"
- ❖ La clé primaire de la table achat est composée par les champs (idClient, idArticle et dateAchat)

La représentation graphique de cette base est :



II. Les anomalies :

1. Absence ou duplication de clés primaires :

Une **clé primaire** est un identifiant unique qui permet de distinguer chaque enregistrement d'une table dans une base de données relationnelle. Elle ne doit **jamais être dupliquée** ou laissée **vide** (valeur nulle).

L'**absence** de clé primaire ou la **duplication** de celle-ci dans une table provoque des **anomalies d'intégrité**. Cela empêche d'identifier de manière unique chaque enregistrement, rendant la manipulation des données incohérente. Ces erreurs peuvent entraîner des **incohérences**, des **problèmes de redondance** et des **difficultés lors des requêtes**.

Exemple1 :

idClient	Nom	Email
1	Ahmed	Ahmed.ah@yahoo.fr
NULL	Layla	Leyla.gz@gmail.com
3	Omar	BejiOmar@outlook.com

Problème :

Ici, un enregistrement (Layla) n'a pas de idClient, ce qui signifie que cet enregistrement ne peut pas être identifié de manière unique. Cela peut poser des problèmes lorsqu'on veut mettre à jour ou supprimer cet enregistrement, car on ne pourra pas le retrouver de manière fiable.

Exemple2:

idClient	Nom	Email
1	Ahmed	ahmed@mail.com
2	Layla	layla@mail.com
2	Omar	omar@mail.com

Problème :

Ici, les deux enregistrements pour Layla et Omar ont le même ClientID = 2. Cela pose un problème, car il devient impossible **de distinguer ces deux enregistrements** lors d'une requête ou d'une manipulation de données.

2. Clés étrangères non conformes

Une **clé étrangère** (ou **foreign key**) est un champ dans une table qui fait référence à la **clé primaire** d'une autre table. Elle établit un lien entre deux tables et garantit l'intégrité référentielle dans une base de données relationnelle.

L'anomalie des **clés étrangères non conformes** se produit lorsque la valeur d'une clé étrangère fait référence à un enregistrement **qui n'existe pas** dans la table référencée (celle contenant la clé primaire). Cela brise l'intégrité référentielle, ce qui peut entraîner des **incohérences dans les données**.

Exemple

Supposons que nous avons deux tables dans une base de données : **Clients** et **Achats**. La table **Achats** contient une clé étrangère qui fait référence à la table **Clients**.

Données de la table Clients :

IdClient	Nom	Email
1	Ahmed	ahmed@mail.com
2	Layla	layla@mail.com
3	Omar	omar@mail.com

Données de la table Achats :

idClient	Date d'achat	Quantité
1	2024-01-15	1
2	2024-01-20	2
4	2024-02-10	1

Problème :

Dans la table **Achats**, la ligne 3 contient une valeur de **idClient = 4**, mais il n'y a **aucun enregistrement** avec **ClientID = 4** dans la table **Clients**. Cela signifie que cet achat fait référence à un client qui **n'existe pas** dans la base de données.

3. Redondance des données :

La **redondance des données** se produit lorsqu'une information est **répétée inutilement** dans une ou plusieurs tables d'une base de données. Cela entraîne une **duplication** des informations, ce qui est inefficace et peut causer des **incohérences** et des **anomalies de mise à jour**.

Les bases de données relationnelles visent à minimiser cette redondance en normalisant les données (en les organisant dans des tables distinctes reliées par des clés primaires et étrangères). Lorsque cette normalisation n'est pas respectée, les mêmes données peuvent apparaître plusieurs fois, ce qui crée des problèmes.

Exemple

Prenons l'exemple d'une base de données qui gère des informations sur des clients, des produits et des achats. Si toutes ces informations sont stockées dans une seule table, cela peut entraîner une redondance.

Table Achats:

NomClient	Email	NomProduit	PrixProduit	DateAchat
Ahmed	ahmed@mail.com	Ordinateur portable	800	2024-01-10
Layla	layla@mail.com	Smartphone	500	2024-01-15
Ahmed	ahmed@mail.com	Smartphone	500	2024-02-05

Problème de redondance :

- Les informations sur Ahmed (son nom et son email) sont répétées dans chaque ligne où il a effectué un achat.
- Chaque fois qu'Ahmed achète un produit, son nom et son email doivent être répétés, ce qui entraîne une duplication des informations.
- Cela peut poser des problèmes lorsque les informations de Ahmed changent (par exemple, s'il change d'email). Il faudra mettre à jour chaque ligne où ses informations apparaissent, ce qui peut entraîner des incohérences si certaines lignes ne sont pas mises à jour correctement.

Solution : Normalisation

Pour résoudre ce problème de redondance, la base de données doit être **normalisée**. La normalisation consiste à décomposer **une grande table en plusieurs tables plus petites** afin de minimiser la redondance.

Dans cet exemple, nous pouvons diviser la table Achats en plusieurs tables : Clients, Produits, et Achats, tout en reliant les tables par des clés étrangères.

Table Clients (normalisée) :

idClient	Nom	Email
1	Ahmed	ahmed@mail.com
2	Layla	layla@mail.com

Table Produits (normalisée) :

idProduit	NomProduit	Prix
101	Ordinateur portable	800
102	Smartphone	500
103	Clavier mécanique	100

Table Achats (normalisée) :

idClient#	idProduit#	Date d'achat
1	101	2024-01-10
2	102	2024-01-15
1	103	2024-02-05

Résultat :

- ❖ Les informations sur les clients sont stockées une seule fois dans la table Clients.
- ❖ Les informations sur les produits sont également stockées une seule fois dans la table Produits.
- ❖ La table Achats ne contient que des références aux idClient et idProduit, sans redondance des données.

4. Violation des contraintes de domaine :

Une **contrainte de domaine** est une règle qui impose des restrictions sur les valeurs pouvant être stockées dans une colonne d'une table. Ces contraintes assurent que les données respectent des critères spécifiques, tels que le type de données, la plage de valeurs, ou des formats spécifiques.

La **violation des contraintes de domaine** se produit lorsqu'une valeur insérée dans une colonne ne respecte pas ces règles. Cela peut entraîner des incohérences dans la base de données et des erreurs lors des requêtes ou des mises à jour.

Exemple

Imaginons une base de données qui gère les informations sur des étudiants et qui contient une table appelée **Étudiants**. Supposons que nous avons les contraintes suivantes pour les colonnes de cette table :

- ❖ **ID** : entier positif.
- ❖ **Âge** : entier devant être compris entre 18 et 50 ans.
- ❖ **Email** : doit respecter un format d'email valide *****@****.*****

ID	Nom	Âge	Email
1	Ahmed	20	ahmed@mail.com
2	Layla	17	layla@mail.com
-3	Omar	30	omar@mail.com
4	Mounir	23	mounir@mail@com

Problèmes de violation des contraintes de domaine :

1. **Âge de Layla :**
 - Layla a un âge de **17**, ce qui ne respecte pas la contrainte d'âge qui exige que les étudiants soient âgés de **18 à 25 ans**.
2. **ID d'Omar :**
 - L'ID de Omar est -3, ne respecte pas la contrainte que l'ID doit être un entier positif.
3. **Email de Mounir :**

- L'email de Zara, **zara@mail@com**, ne respecte pas le format d'un email valide (il contient un @ supplémentaire).

III. Manipulations sur la structure d'une base de données (Définition des données) :

1. Langage SQL :

SQL (Structured Query Language) est un langage standard utilisé pour interagir avec les bases de données relationnelles. Il permet d'effectuer diverses opérations comme la création, la modification, la gestion et la requête des données stockées dans des bases de données sous forme de tables.

2. Définition des bases :

La commande **CREATE DATABASE** est utilisée pour **créer** une nouvelle base de données dans un système de gestion de bases de données (SGBD) comme MySQL

Exemple :

Si vous souhaitez créer une base de données nommée **Ecole**, la requête sera :

```
CREATE DATABASE Ecole;
```

La commande **DROP DATABASE** est utilisée pour **supprimer** une base de données entière, ainsi que toutes les tables et données qu'elle contient. Cette opération est **irréversible** et doit être utilisée avec précaution, car elle supprime définitivement toutes les données de la base de données.

Exemple :

Si vous voulez supprimer la base de données **Ecole**, la commande sera :

```
DROP DATABASE Ecole;
```

3. Définition des tables :

a. Les types des données en SQL :

Type SQL	Description	Exemple d'utilisation
INT	Entier sur 4 octets (32 bits), utilisé pour stocker des nombres entiers.	INT : Stocke des valeurs entières comme 100, -50, 2023. Exemple : AGE INT;
DECIMAL	Nombre décimal avec une précision fixe (nombre total de chiffres) et une échelle (chiffres après la virgule).	DECIMAL(10, 2) : Nombre avec jusqu'à 10 chiffres au total, dont 2 après la virgule. Exemple : PRIX DECIMAL(10, 2);
CHAR	Chaîne de caractères de longueur fixe. Si la longueur n'est pas atteinte, la chaîne est complétée par des espaces.	CHAR(10) : Chaîne fixe de 10 caractères. Exemple : CODE CHAR(10); pour des codes de produit de longueur fixe.
VARCHAR	Chaîne de caractères de longueur variable. La taille est flexible jusqu'à une limite définie.	VARCHAR(50) : Chaîne de longueur variable jusqu'à 50 caractères. Exemple : NOM VARCHAR(50); pour stocker des noms.

Type SQL	Description	Exemple d'utilisation
TEXT	Utilisé pour de longues chaînes de caractères, comme des paragraphes ou des descriptions.	TEXT : Texte long sans limite de taille prédéfinie. Exemple : DESCRIPTION TEXT; pour des descriptions détaillées.
DATE	Représente une date au format AAAA-MM-JJ (année-mois-jour).	DATE : Stocke une date. Exemple : DATE_NAISSANCE DATE; avec une valeur comme 1990-05-12.
TIME	Représente une heure au format HH:MM (heure:minute).	TIME : Stocke une heure. Exemple : HEURE_ENTRÉE TIME; avec une valeur comme 14:30:00.
DATETIME	Combine une date et une heure au format AAAA-MM-JJ HH:MM	DATETIME : Stocke une date et une heure. Exemple : CREATION DATETIME; avec une valeur comme 2023-12-01 14:30:00.

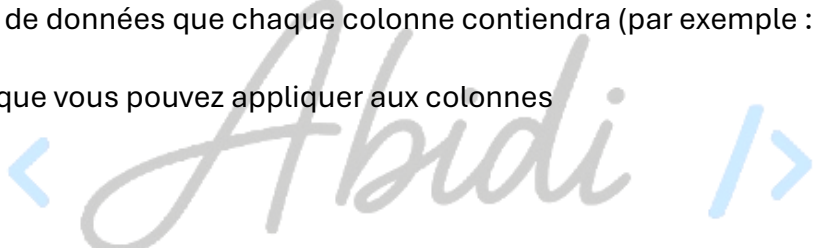
b. Création d'une table :

La commande **CREATE TABLE** en SQL est utilisée pour créer une nouvelle table dans une base de données. Elle permet de définir la structure de la table en spécifiant les colonnes, leurs types de données et d'autres contraintes comme les clés primaires, les clés étrangères, les valeurs par défaut, etc.

Syntaxe de base :

```
CREATE TABLE NomTable (
    Colonne1 TypeDeDonnée(taille) [Contraintes],
    Colonne2 TypeDeDonnée(taille) [Contraintes],
    , [contraintes définition complète]
);
```

- ❖ **NomTable** : Le nom que vous donnez à la table.
- ❖ **Colonne1, Colonne2** : Les noms des colonnes que vous créez.
- ❖ **TypeDeDonnée** : Le type de données que chaque colonne contiendra (par exemple : INT, VARCHAR, DATE).
- ❖ **Contraintes** : Les règles que vous pouvez appliquer aux colonnes



 LYCÉE MONJI

 SLIM SBIBA

Liste des contraintes :

Contrainte	Description	Exemple d'utilisation
PRIMARY KEY	Assure que chaque enregistrement dans une table est unique et non nul.	ClientID INT PRIMARY KEY
NOT NULL	Empêche une colonne de contenir des valeurs nulles (obligatoire).	Nom VARCHAR(50) NOT NULL
UNIQUE	Assure que toutes les valeurs d'une colonne (ou d'un ensemble de colonnes) sont uniques dans la table.	Email VARCHAR(100) UNIQUE
DEFAULT	Définit une valeur par défaut pour une colonne si aucune valeur n'est spécifiée lors de l'insertion.	DateInscription DATE DEFAULT "1990-01-01"
REFERENCES	Déclare une clé étrangère (foreign key) qui fait référence à une colonne d'une autre table pour garantir l'intégrité référentielle.	ClientID INT REFERENCES Client(ClientID);
AUTO_INCREMENT	Incrémente automatiquement la valeur à chaque nouvelle insertion. S'applique à une colonne de type entier (INT)	id INT AUTO_INCREMENT PRIMARY KEY

c. Contraintes définition complète :**c.1. FOREIGN KEY**

- ❖ La contrainte **FOREIGN KEY** (clé étrangère) est utilisée pour lier deux tables.
- ❖ Elle fait référence à une colonne ou à un ensemble de colonnes dans une autre table, généralement à une **PRIMARY KEY**.
- ❖ Elle garantit l'**intégrité référentielle** entre les tables. Cela signifie que la valeur de la clé étrangère dans la table enfant doit exister dans la table parent.

Syntaxe dans CREATE TABLE :

```
CREATE TABLE TableEnfant (
    Colonne1 TypeDeDonnée,
    ColonneFK TypeDeDonnée,
    FOREIGN KEY (ColonneFK) REFERENCES TableParent(ColonneParent)
);
```

Exemple :

```
CREATE TABLE Commande (
    CommandeID INT PRIMARY KEY,
    ClientID INT,
    DateCommande DATE,
    FOREIGN KEY (ClientID) REFERENCES Client(ClientID) );
```


Dans cet exemple, **ClientID** dans la table **Commande** est une **clé étrangère** qui fait référence à la colonne **ClientID** de la table **Client**. Cela garantit qu'une commande ne peut être liée qu'à un client existant.

Remarques :

Les clauses **ON UPDATE CASCADE** et **ON DELETE CASCADE** sont utilisées en conjonction avec les **contraintes de clé étrangère (FOREIGN KEY)** dans SQL. Elles définissent le comportement à adopter lorsqu'une ligne dans la table parent est mise à jour ou supprimée, en synchronisant automatiquement les modifications dans la table enfant.

Exemple :

```
CREATE TABLE Clients (
    ClientID INT PRIMARY KEY,
    Nom VARCHAR(100)
);
CREATE TABLE Commandes (
    CommandeID INT PRIMARY KEY,
    ClientID INT,
    FOREIGN KEY (ClientID) REFERENCES Clients(ClientID)
    ON UPDATE CASCADE
    ON DELETE CASCADE
);
```

Explication :

- **ON UPDATE CASCADE** : Si l'ID d'un client dans **Clients** est mis à jour, la colonne **ClientID** dans **Commandes** sera automatiquement mise à jour.
- **ON DELETE CASCADE** : Si un client est supprimé de **Clients**, toutes ses commandes seront également supprimées de **Commandes**.

c.2. CHECK

- ❖ La contrainte **CHECK** impose une condition qui doit être vraie pour chaque enregistrement dans la colonne ou dans l'ensemble de colonnes spécifié.
- ❖ Elle permet de restreindre les valeurs qu'une colonne peut accepter.

Syntaxe dans CREATE TABLE

```
CREATE TABLE TableName (
    Colonne1 TypeDeDonnée CHECK (Condition),
    Colonne2 TypeDeDonnée,
    ...
);
```

Exemple :

```
CREATE TABLE Produit (
    ProduitID INT PRIMARY KEY,
    Nom VARCHAR(100) NOT NULL,
    Prix DECIMAL(10, 2),
    QuantitéStock INT CHECK (QuantitéStock >= 0)
-- QuantitéStock doit être supérieure ou égale à 0
);
```

Liste opérateurs de comparaison SQL

Opérateur	Description	Exemple SQL avec CHECK	Explication de l'exemple
=	Vérifie l'égalité entre deux valeurs	CHECK (Age = 30)	Restreint l'âge à la valeur exacte de 30.
<> ou !=	Vérifie l'inégalité (différent de)	CHECK (Age <> 30)	Interdit les enregistrements où l'âge est exactement 30.
>	Vérifie si la valeur de gauche est supérieure à celle de droite	CHECK (Age > 18)	Permet uniquement des âges supérieurs à 18.
<	Vérifie si la valeur de gauche est inférieure à celle de droite	CHECK (Age < 65)	Permet uniquement des âges inférieurs à 65.
>=	Vérifie si la valeur de gauche est supérieure ou égale à celle de droite	CHECK (Salaire >= 3000)	Garantit que le salaire est d'au moins 3000.
<=	Vérifie si la valeur de gauche est inférieure ou égale à celle de droite	CHECK (Salaire <= 10000)	Garantit que le salaire ne dépasse pas 10 000.
BETWEEN	Vérifie si une valeur est comprise entre deux valeurs	CHECK (Age BETWEEN 18 AND 65)	Restreint l'âge à être entre 18 et 65 inclus.
IN	Vérifie si une valeur est présente dans une liste de valeurs	CHECK (Statut IN ('Actif', 'Inactif'))	Limite le statut aux valeurs "Actif" ou "Inactif".

Opérateurs logiques SQL

Opérateur	Description	Exemple	Explication de l'exemple
AND	Renvoie vrai si toutes les conditions sont vraies	CHECK (Age >= 18 AND Age <= 65)	Limite l'âge à être entre 18 et 65.
OR	Renvoie vrai si au moins une condition est vraie	CHECK (Sexe = 'F' OR Sexe = 'M')	Permet que le sexe soit soit "F" soit "M".
NOT	Renverse le résultat d'une condition	CHECK (NOT (Age < 18))	Interdit les enregistrements où l'âge est inférieur à 18.

c.3. Clé primaire

Une **clé primaire composée** est une clé primaire qui est définie sur plusieurs colonnes (ou champs) d'une table. Cela signifie que la combinaison de ces colonnes doit être **unique** et **non nulle** pour chaque enregistrement, mais aucune de ces colonnes ne peut, à elle seule, identifier de manière unique une ligne dans la table.

Pourquoi utiliser une clé primaire composée ?

- Une clé primaire composée est utilisée lorsque plusieurs colonnes ensemble définissent de manière unique un enregistrement, mais qu'aucune colonne prise individuellement ne le fait.
- C'est souvent utilisé dans les **tables de relation** pour modéliser des associations entre plusieurs tables.

Syntaxe de la clé primaire composée dans une requête CREATE TABLE :

```
CREATE TABLE NomTable (
    Colonne1 TypeDeDonnée,
    Colonne2 TypeDeDonnée,
    ...,
    PRIMARY KEY (Colonne1, Colonne2)
);
```

Dans cet exemple, la clé primaire est définie sur plusieurs colonnes (**Colonne1** et **Colonne2**), et la combinaison de leurs valeurs doit être unique dans la table.

Exemple : Table Inscription

Imaginons une table **Inscription** qui enregistre l'association entre des étudiants et des cours. Chaque étudiant peut suivre plusieurs cours, et chaque cours peut être suivi par plusieurs étudiants. Pour garantir qu'un étudiant ne s'inscrit pas plusieurs fois au même cours, nous allons utiliser une clé primaire composée des colonnes **EtudiantID** et **CoursID**.

Requête :

```
CREATE TABLE Inscription (
    EtudiantID INT,
    CoursID INT,
    DateInscription DATE,
    PRIMARY KEY (EtudiantID, CoursID),
    FOREIGN KEY (EtudiantID) REFERENCES Etudiant(EtudiantID),
    FOREIGN KEY (CoursID) REFERENCES Cours(CoursID)
);
```

d. Supprimer une table :

La requête SQL DROP TABLE est utilisée pour **supprimer** une table existante dans une base de données, ainsi que toutes les données qu'elle contient. Une fois exécutée, cette opération est **irréversible**, c'est-à-dire que la table et ses données sont définitivement supprimées.

Syntaxe :

```
DROP TABLE NomTable;
```

Exemple :

Imaginons que nous avons une table appelée **Client** dans notre base de données, et que nous souhaitons la supprimer :

```
DROP TABLE Client;
```

Cette commande supprime la table **Client** ainsi que toutes les données et les contraintes associées à cette table.

Remarque :

Si la table a des relations (clés étrangères) avec d'autres tables, il peut être nécessaire de désactiver ou de supprimer ces relations avant d'exécuter la commande DROP TABLE.

4. Définition des colonnes :

La commande SQL **ALTER TABLE** permet de modifier la structure d'une table existante dans une base de données. Cela inclut l'ajout, la suppression, la modification, ou le renommage de colonnes. Voici une explication de ces fonctionnalités, accompagnée d'exemples :

Opération	Syntaxe	Exemple
Ajouter une colonne	ALTER TABLE NomTable ADD COLUMN NomColonne TypeDeDonnée;	ALTER TABLE Client ADD COLUMN Email VARCHAR(100);
Supprimer une colonne	ALTER TABLE NomTable DROP COLUMN NomColonne;	ALTER TABLE Client DROP COLUMN Adresse;
Modifier une colonne	ALTER TABLE NomTable ALTER COLUMN NomColonne NouveauType;	ALTER TABLE Client ALTER COLUMN Age VARCHAR(3);
Renommer une colonne	ALTER TABLE NomTable RENAME COLUMN AncienNom TO NouveauNom;	ALTER TABLE Client RENAME COLUMN Nom TO NomClient;

5. Définitions des contraintes :

La commande SQL **ALTER TABLE** permet de modifier la structure d'une table en ajoutant ou supprimant des contraintes, ou en activant/désactivant des contraintes existantes. Voici une explication de ces opérations, avec des exemples concrets :

Opération	Syntaxe	Exemple
Ajouter une contrainte	ALTER TABLE NomTable ADD CONSTRAINT NomContrainte TypeDeContrainte (NomColonne);	ALTER TABLE Commandes ADD CONSTRAINT fk_client FOREIGN KEY (ClientID) REFERENCES Clients (ClientID);
Supprimer une contrainte	ALTER TABLE NomTable DROP CONSTRAINT NomContrainte;	ALTER TABLE Commandes DROP CONSTRAINT fk_client;
Activer une contrainte	ALTER TABLE NomTable ENABLE CONSTRAINT NomContrainte;	ALTER TABLE Commandes ENABLE CONSTRAINT fk_client;
Désactiver une contrainte	ALTER TABLE NomTable DISABLE CONSTRAINT NomContrainte;	ALTER TABLE Commandes DISABLE CONSTRAINT fk_client;

Types de contraintes courantes :

1. **PRIMARY KEY** : Un identifiant unique pour chaque ligne de la table.
2. **FOREIGN KEY** : Crée un lien entre deux tables.
3. **CHECK** : Impose une règle sur les valeurs qu'une colonne peut accepter.

IV. Manipulations sur les données d'une base de données (Manipulations des données) :

1. Insertion des données :

La commande SQL **INSERT INTO** est utilisée pour insérer des nouvelles données dans une table d'une base de données. Vous pouvez insérer des données dans toutes les colonnes d'une table ou seulement dans certaines colonnes spécifiques.

Action	Syntaxe	Exemple
Insérer dans toutes les colonnes	INSERT INTO NomTable VALUES (valeur1, valeur2, ..., valeurN);	INSERT INTO Clients VALUES (1, 'Ahmed', 'ahmed@example.com');
Insérer dans certaines colonnes	INSERT INTO NomTable (Colonne1, Colonne2, ...) VALUES (valeur1, valeur2, ...);	INSERT INTO Clients (Nom, Email) VALUES ('Layla', 'layla@example.com');
Insérer plusieurs lignes	INSERT INTO NomTable (Colonne1, Colonne2, ...) VALUES (valeur1, valeur2), (valeur1, ...), ...	INSERT INTO Clients (Nom, Email) VALUES ('Ahmed', 'ahmed@example.com'), ('Layla', 'layla@example.com');

2. Suppression des données :

La commande **DELETE** permet de supprimer des enregistrements dans une table. Elle peut être utilisée pour supprimer toutes les lignes d'une table ou seulement certaines lignes, en fonction d'une condition spécifiée dans la clause **WHERE**.

Action	Syntaxe	Exemple
Suppression simple	DELETE FROM NomTable WHERE condition;	DELETE FROM Clients WHERE ClientID = 1;
Suppression de tous les enregistrements	DELETE FROM NomTable;	DELETE FROM Clients;

3. Modifications des données :

La requête **UPDATE** permet de modifier les valeurs d'une table en fonction de conditions spécifiques. Sans jointure, elle ne concerne qu'une seule table.

Action	Syntaxe	Exemple
Mettre à jour une valeur	UPDATE table_name SET column_name = new_value WHERE condition;	<i>Modifie le salaire de l'employé dont l'ID est 1.</i> UPDATE Employes SET salaire = 4000 WHERE employe_id = 1;
Mettre à jour plusieurs colonnes	UPDATE table_name SET column1 = new_value1, column2 = new_value2 WHERE condition;	<i>Met à jour le salaire et le département de l'employé avec l'ID 3.</i> UPDATE Employes SET salaire = 4500 ,departement = 'Marketing' WHERE employe_id = 3;
Mettre à jour toutes les lignes (sans WHERE)	UPDATE table_name SET column_name = new_value;	<i>Modifie le salaire de tous les employés dans la table.</i> UPDATE Employes SET salaire = 5000;
Modifier une valeur existante	UPDATE table_name SET column_name = column_name + increment_value WHERE condition;	<i>Augmente le salaire des employés du département IT de 500.</i> UPDATE Employes SET salaire = salaire + 500 WHERE departement = 'IT';

4. Sélection des données

La requête **SELECT** en SQL permet de **recupérer des données** dans une table de base de données. C'est l'une des opérations les plus basiques et les plus importantes en SQL.

Structure de base d'une requête **SELECT**

```
SELECT colonnes
FROM table;
```

❖ **SELECT**

- Cela signifie : **choisir** les colonnes que tu veux afficher.
- Par exemple : SELECT nom, age signifie que tu veux récupérer les colonnes nom et age.
- SELECT * signifie sélectionner tous les colonnes.

❖ FROM

- C'est là où tu indiques **de quelle table(s)** tu veux récupérer les données. Par exemple : FROM utilisateurs.

Exemple:

Si tu as une table appelée utilisateurs avec ces données :

id	nom	age
1	Alice	25
2	Bob	30
3	Carol	22

Une requête pour afficher les noms et âges de tous les utilisateurs ressemble à ça :

```
SELECT nom, age
FROM utilisateurs;
```

Résultat :

nom	age
Alice	25
Bob	30
Carol	22

a. Filtrage des résultats :

La clause **WHERE** en SQL est utilisée pour filtrer les enregistrements dans une requête **SELECT** en fonction d'une ou plusieurs conditions. Elle permet de récupérer **uniquement** les lignes qui répondent aux **critères spécifiés**.

Syntaxe générale

```
SELECT colonne1, colonne2, ...
FROM table
WHERE condition;
```

LYCÉE MONJI
SLIM SBIBA

Fonctionnement de la clause WHERE

- **SELECT** : Utilisé pour spécifier les colonnes que vous souhaitez récupérer.

- **FROM** : Indiquer la(les) table(s) à partir de laquelle les données sont extraites.
- **WHERE** : Spécifie les conditions que les lignes doivent respecter pour être incluses dans les résultats.

Exemple1 :

Supposons que nous ayons une table employes comme ci-dessous :

id_employe	nom	age	salaire
1	Alice	30	50000
2	Bob	45	60000
3	Charlie	25	40000
4	David	35	55000

Si vous souhaitez récupérer les informations des employés qui ont un salaire supérieur à 50 000

```
SELECT nom, salaire
FROM employes
WHERE salaire > 50000;
```

Résultat :

nom	salaire
Bob	60000
David	55000

Ici, la condition `salaire > 50000` filtre les lignes pour ne conserver que celles où le salaire est supérieur à 50 000.

Exemple 2 : Utilisation avec plusieurs conditions (opérateur AND)

Si vous souhaitez récupérer les informations des employés âgés de plus de 30 ans et ayant un salaire supérieur à 50 000 :

```
SELECT nom, age, salaire
FROM employes
WHERE age > 30 AND salaire > 50000;
```

Résultat :

nom	age	salaire
Bob	45	60000
David	35	55000

Ici, les deux conditions doivent être vraies (`age > 30` **et** `salaire > 50000`) pour que l'enregistrement soit inclus dans le résultat.

Exemple 3 : Utilisation de l'opérateur OR

Si vous souhaitez récupérer les informations des employés qui ont soit plus de 30 ans soit un salaire supérieur à 50 000 :

```
SELECT nom, age, salaire
FROM employes
WHERE age > 30 OR salaire > 50000;
```

Résultat :

nom	age	salaire
Alice	30	50000
Bob	45	60000
David	35	55000

Avec l'opérateur OR, il suffit qu'une seule des conditions soit vraie pour que la ligne soit incluse.

Exemple 4 : Utilisation de l'opérateur LIKE

Si vous souhaitez récupérer les noms des employés dont le nom commence par "A" :

```
SELECT nom
FROM employes
WHERE nom LIKE 'A%';
```

Résultat :

nom
Alice

Ici, l'opérateur **LIKE** est utilisé pour rechercher des correspondances partielles. 'A%' signifie "commence par A".

Voici un tableau des jokers principaux utilisés avec l'opérateur LIKE en SQL

Joker	Description	Exemple de motif	Explication du motif
%	Représente zéro, un ou plusieurs caractères	C%	Correspond à toute chaîne commençant par "C"
_	Représente exactement un caractère	C_____	Correspond à toute chaîne commençant par "C" suivie de 7 autres caractères

Exemple 5 : IS NULL IS NOT NULL :

En SQL, les opérateurs `IS NULL` et `IS NOT NULL` sont utilisés pour vérifier la présence ou l'absence de valeurs `NULL` dans une colonne. Voici leurs fonctions et des exemples d'utilisation.

Soit table appelée employes suivant :

id_employe	nom	age		salaire
1	Alice	30		50000
2	Bob	NULL		60000
3	Charlie	25		NULL
4	David	35		55000

Requête avec IS NULL :

```
SELECT nom, age
FROM employes
WHERE age IS NULL;
```

Résultat :

nom	age
Bob	NULL

Requête avec IS NOT NULL :

```
SELECT nom, salaire
FROM employes
WHERE salaire IS NOT NULL;
```

Résultat :

nom	salaire
Alice	50000
Bob	60000
David	55000

b. Jointure :

En SQL, une **jointure** est utilisée pour combiner des lignes provenant de deux ou plusieurs tables en fonction d'une condition qui lie les tables, souvent basée sur une relation entre une clé primaire et une clé étrangère. Pour faire la jointure les tables sont listées dans la clause `FROM`, et la condition de jointure est définie dans la clause `WHERE`.

Syntaxe :

```
SELECT Colonnes
FROM Table1, Table2
WHERE Table1.Colonne = Table2.Colonne;
```

Exemple :

Soient deux tables : **Clients** et **Commandes**. La table **Clients** contient les informations des clients, et la table **Commandes** contient les commandes effectuées par ces clients. Les deux tables sont reliées par la colonne **ClientID** (la clé primaire dans **Clients** et la clé étrangère dans **Commandes**).

Clients :

ClientID	Nom
1	Ahmed
2	Layla

Commandes :

CommandeID	ClientID	Produit
101	1	Ordinateur
102	2	Smartphone
103	1	Imprimante

Requête avec jointure:

```
SELECT A.Nom AS Client, C.Produit
FROM Clients AS A, Commandes AS C
WHERE A.ClientID = C.ClientID;
```

Résultat :

Client	Produit
Ahmed	Ordinateur
Layla	Smartphone
Ahmed	Imprimante

Explication :

- ✓ La requête sélectionne les noms des clients dans la table **Clients** et les produits commandés dans la table **Commandes**.
- ✓ La condition **WHERE Clients.ClientID = Commandes.ClientID** relie les deux tables en fonction du **ClientID**.
- ✓ Le résultat montre les produits commandés par chaque client.
- ✓ L'opérateur **AS** permet de créer des alias

c. Fonctions sur les types DATE en SQL

SQL fournit plusieurs fonctions pour manipuler les dates, permettant d'extraire des informations spécifiques comme le jour, le mois, l'année, ou encore la date actuelle. Voici un aperçu des fonctions couramment utilisées sur les types **DATE**.

Liste des fonctions :

Fonction	Description	Exemple de requête	Résultat
DAY()	Renvoie le jour d'une date.	SELECT DAY('2024-10-06');	6
MONTH()	Renvoie le mois d'une date.	SELECT MONTH('2024-10-06');	10
YEAR()	Renvoie l'année d'une date.	SELECT YEAR('2024-10-06');	2024
NOW()	Renvoie la date et l'heure actuelles.	SELECT NOW();	2025-07-03 14:30:15
DATE()	Renvoie la partie date (sans l'heure).	SELECT DATE(NOW());	2024-10-06
ADDDATE()	Ajoute un intervalle de temps (jours, mois, etc.) à une date donnée	SELECT ADDDATE("2025-10-10", 15)	2025-10-25
ADDDATE(date, INTERVAL valeur unité)	<ul style="list-style-type: none"> ✓ valeur : un nombre entier (positif ou négatif). ✓ unité : l'unité de temps (par exemple : DAY, MONTH, YEAR) 	SELECT ADDDATE("2025-10-10", INTERVAL 3 MONTH)	2026-01-10
DATEDIFF()	Calcule la différence (en jours) entre deux dates. Il retourne un entier : $DATEDIFF(date1, date2) = date1 - date2$	SELECT DATEDIFF('2025-07-15', '2025-07-01')	14

d. Fonctions d'agrégation en SQL

Les **fonctions d'agrégation** en SQL sont utilisées pour effectuer des calculs sur un ensemble de lignes et renvoyer une valeur unique.

Voici les principales fonctions d'agrégation en SQL

Fonction	Description	Exemple de requête	Résultat
COUNT()	Compte le nombre de lignes.	SELECT COUNT(*) FROM Commandes;	Nombre total de lignes
SUM()	Calcule la somme d'une colonne numérique.	SELECT SUM(Prix) FROM Commandes;	Somme des prix

Fonction	Description	Exemple de requête	Résultat
AVG()	Calcule la moyenne d'une colonne numérique.	SELECT AVG(Prix) FROM Commandes;	Moyenne des prix
MAX()	Renvoie la valeur maximale d'une colonne.	SELECT MAX(Prix) FROM Commandes;	Prix maximum
MIN()	Renvoie la valeur minimale d'une colonne.	SELECT MIN(Prix) FROM Commandes;	Prix minimum

e. Groupement en SQL

Le **groupement** en SQL est utilisé pour regrouper les lignes d'une table qui partagent des valeurs communes dans une ou plusieurs colonnes, puis appliquer des **fonctions d'agrégation** (comme COUNT(), SUM(), AVG(), etc.) sur chaque groupe. La clause utilisée pour effectuer ce groupement est **GROUP BY**.

Syntaxe de base :

```
SELECT colonne1, fonction_agrégation(colonne2)
FROM table
GROUP BY colonne1;
```

Fonctionnement du groupement avec GROUP BY :

1. **Regroupement des données** : Le **GROUP BY** regroupe les lignes ayant des valeurs identiques dans les colonnes spécifiées.
2. **Application des fonctions d'agrégation** : Les fonctions d'agrégation sont appliquées aux groupes formés.
3. **Obtenir un résultat pour chaque groupe** : La requête renvoie une seule ligne par groupe avec les résultats des fonctions d'agrégation.

Exemple simple

Supposons qu'on ait une table **Ventes** avec les données suivant :

id	Produit	Quantite	Prix
1	Ordinateur	1	1200
2	Souris	3	15
3	Clavier	2	50
4	Ordinateur	1	1200
5	Souris	2	15
6	Casque Audio	1	30
7	Clavier	1	50

On souhaite connaître la **quantité totale vendue par produit**.

```
SELECT Produit, SUM(Quantite) AS QuantiteTotale
FROM Ventes
GROUP BY Produit;
```

Résultat :

Produit	QuantiteTotale
Ordinateur	2
Souris	5
Clavier	3
Casque Audio	1

Utilisation de HAVING avec GROUP BY

La clause **HAVING** est utilisée pour filtrer les résultats après le **groupe**ment. Contrairement à **WHERE**, qui filtre les lignes avant le groupement, **HAVING** filtre les groupes une fois qu'ils sont formés.

Exemple avec HAVING

Supposons que nous voulons obtenir uniquement les produits dont la **quantité totale vendue** est supérieure ou égal à 3.

```
SELECT Produit, SUM(Quantite) AS QuantiteTotale
FROM Ventes
GROUP BY Produit
HAVING SUM(Quantite) >= 3;
```

Résultat :

Produit	QuantiteTotale
Souris	5
Clavier	3

Groupement sur plusieurs colonnes

Il est possible de faire un **groupement sur plusieurs colonnes**. Cela permet de créer des sous-groupes selon plusieurs critères.

Exemple

Une entreprise de livraison stocke les **commandes livrées** dans une table Livraisons. Chaque livraison est associée à un **gouvernorat**, une **ville**, et une **quantité de colis livrés**.

id	gouvernorat	ville	nb_colis
1	Tunis	La Marsa	10
2	Tunis	Ariana Ville	8
3	Sfax	Sfax Ville	15
4	Sfax	Sakiet Ezzit	7
5	Tunis	La Marsa	5
6	Sfax	Sfax Ville	12
7	Tunis	Ariana Ville	4

Soit la requête SQL suivante :

```
SELECT gouvernorat, ville, SUM(nb_colis) AS total_colis
FROM Livraisons
GROUP BY gouvernorat, ville;
```

Résultat :

gouvernorat	ville	total_colis
Tunis	La Marsa	15
Tunis	Ariana Ville	12
Sfax	Sfax Ville	27
Sfax	Sakiet Ezzit	7

f. Tri en SQL

La clause **ORDER BY** en SQL permet de trier les résultats d'une requête **par ordre croissant ou décroissant** en fonction d'une ou plusieurs colonnes. Par défaut, les résultats sont triés en **ordre croissant**. On peut également spécifier un **ordre décroissant** à l'aide de la commande **DESC**.

Syntaxe de base :

```
SELECT colonne1, colonne2, ...
FROM table
ORDER BY colonne1 [ASC|DESC], colonne2 [ASC|DESC];
```

- **ASC** : Trie par ordre croissant (par défaut, si aucun ordre n'est spécifié).
- **DESC** : Trie par ordre décroissant.

Exemples d'utilisation

Exemple1 : Trier par une seule colonne (ordre croissant)

Supposons que nous avons une table **Produits** suivante :

refProduit	nomProduit	quantité	prix
P001	Ordinateur Portable	10	1500.00
P002	Souris Sans Fil	50	25.00
P003	Clavier Mécanique	30	70.00
P004	Écran 24 pouces	20	300.00
P005	Disque SSD 1To	15	120.00

Nous voulons obtenir les produits **triés par prix** (ordre croissant).

```
SELECT NomProduit, Prix
FROM Produits
ORDER BY Prix;
```

Résultat :

NomProduit	Prix
Souris Sans Fil	25.00
Clavier Mécanique	70.00
Disque SSD 1To	120.00
Écran 24 pouces	300.00
Ordinateur Portable	1500.00

Exemple 2 : Trier par numéro de colonne

Si nous voulons obtenir la même liste **triée par prix décroissant**, on utilise **DESC**.

```
SELECT NomProduit, Prix
FROM Produits
ORDER BY 2 DESC;
```

Résultat :

NomProduit	Prix
Ordinateur Portable	1500.00
Écran 24 pouces	300.00
Disque SSD 1To	120.00
Clavier Mécanique	70.00
Souris Sans Fil	25.00

Exemple 3 : Trier par plusieurs colonnes

On peut trier les résultats par plusieurs colonnes. Par exemple, si nous voulons **trier les produits par prix croissant** et, en cas d'égalité, **trier par nom de produit en ordre alphabétique**.

Soit la table livraisons suivante :

id	gouvernorat	ville	nb_colis
1	Tunis	La Marsa	10
2	Tunis	Ariana Ville	8
3	Sfax	Sfax Ville	15
4	Sfax	Sakiet Ezzit	7
5	Tunis	La Marsa	5
6	Sfax	Sfax Ville	12
7	Tunis	Ariana Ville	4

Soit la requête SQL permettant de lister les gouvernorats et les villes (tri par **gouvernorat (A→Z)** puis par **ville (A→Z)**)

```
SELECT gouvernorat, ville, nb_colis
FROM Livraisons
ORDER BY gouvernorat ASC, ville ASC;
```

Résultat :

gouvernorat	ville	nb_colis
Sfax	Sakiet Ezzit	7
Sfax	Sfax Ville	15
Sfax	Sfax Ville	12
Tunis	Ariana Ville	8
Tunis	Ariana Ville	4
Tunis	La Marsa	10
Tunis	La Marsa	5

g. Sous-requête :

Une sous-requête est une requête imbriquée utiliser dans la clause **WHERE** pour filtrer les résultats de la requête principale en fonction du résultat d'une autre requête. Cela permet de traiter des conditions complexes dans le filtrage des données.

Syntaxe de base :

```
SELECT colonne1, colonne2, ...
FROM table
WHERE colonne opérateur (sous-requête);
```

- La **sous-requête** retourne une valeur ou un ensemble de valeurs que la requête principale utilise pour filtrer les résultats.
- Les **opérateurs** couramment utilisés avec les sous-requêtes sont des opérateurs de comparaison =, **IN**, **NOT IN**, >, <, etc.

Exemple :Sous-requête avec l'opérateur =

Trouver les employés dont le **salaire** est égal au **salaire minimum** de la table **employées** suivante :

ID	Nom	Salaire
1	Amal	1200
2	Karim	1800
3	Sana	1200
4	Mehdi	2000

Requête :

```
SELECT Nom, Salaire
FROM Employes
WHERE Salaire = (SELECT MIN(Salaire) FROM Salaries);
```

Résultat :

Nom	Salaire
Amal	1200
Sana	1200

Explication : La sous-requête (**SELECT MIN(Salaire) FROM Salaries**) renvoie le salaire minimum, et la requête principale sélectionne les employés ayant ce salaire.

Exemple 2 Sous-requête avec l'opérateur IN

Trouver les produits qui ont été **vendus en 2023**.

Données de la table Produits :

IDProduit	NomProduit
P001	Ordinateur
P002	Souris
P003	Clavier
P004	Écran

Données de la table Ventes :

IDVente	IDProduit	Annee
V01	P001	2023
V02	P003	2023
V03	P004	2022

Requête :

```
SELECT NomProduit
FROM Produits
WHERE IDProduit IN (SELECT IDProduit FROM Ventes WHERE Annee = 2023);
```

Résultat :

NomProduit
Ordinateur
Clavier

Explication : La sous-requête (**SELECT IDProduit FROM Ventes WHERE Annee = 2023**) renvoie les identifiants des produits vendus en 2023. La requête principale sélectionne uniquement les produits dont l'ID se trouve dans cette liste.

