CHAPITRE 4

# LES BASES DE DONNEES
## PASSAGE À LA PRATIQUE : SQLITE

## Sommaire

## 1. Introduction

Dans ce chapitre, vous apprendrez à manipuler une base de données relationnelle (BDR) via le langage de programmation Python et le module SQLite, système de gestion de bases de données relationnelles.

**SQLite :** est une bibliothèque qui propose un moteur de base de données relationnelle (un composant logiciel qui contrôle, lit, enregistre et trie des informations dans une base de données) accessible par le langage ***SQL*** (Structured Query language).

## 2. Installation

Contrairement aux serveurs de bases de données traditionnels, comme MySQL, la particularité de SQLite est de ne pas reproduire le schéma habituel client-serveur (qui nécessite une installation, configuration, …) mais d'être directement intégré aux programmes.

Afin de manipuler une base de données à travers un script Python, on doit importer le module ***sqlite3*** pour y accéder à ses fonctionnalités :

```
>>> import sqlite3
```

SQLite3 peut être intégré à Python en utilisant le module sqlite3. Vous n'avez pas besoin d'installer ce module séparément car il est livré par défaut avec Python version 2.5.x et les versions ultérieures.

## 3. Création d'une base de données

D'abord créer un objet de connexion représentant la base de données :

```
>>> connexion = sqlite3.connect ('nom_base_de_donnees.db')
```

L'objet ***connexion***, assure l'interface entre notre programme et la base de données. À la suite de cette instruction, si la base n'existe pas encore, elle sera créée, et si elle existe déjà elle sera réutilisée.

L'objet ***connexion*** est en place, pour dialoguer avec lui, il faut mettre en place, encore un autre objet interface que l'on appelle un ***curseur***. Il s'agit d'une sorte de tampon mémoire intermédiaire, destiné à mémoriser temporairement les données en cours de traitement, ainsi que les opérations que vous effectuez sur elles, avant leur transfert définitif à la base de données.
La syntaxe est la suivante :

```
>>> cur = connexion.cursor()
```

Cette technique permet donc d'annuler si nécessaire une ou plusieurs opérations inadéquates, et de revenir en arrière dans le traitement, sans que la base de données n'en soit affectée.

Ensuite vous pouvez passer la ***requête*** à exécuter au *curseur* par l'intermédiaire de sa méthode ***execute()***, sous la forme d'un argument de type chaine de caractères.

La syntaxe est la suivante :

```
>>> cur.execute(requete_sql)
```

**Attention :** Quand on exécute des requêtes elles seront mises dans le tampon du curseur, mais elles n'ont pas encore été transférées véritablement dans la base de données. On peut donc annuler tout.

Le transfert définitif dans la base de données sera déclenché par la méthode ***commit()*** de l'objet *connexion* :

```
>>> connexion.commit()
```

A la fin de chaque utilisation, il faut fermer la connexion vers la base :

```
>>> connexion.close()
```

## 4. Manipulation des tables

### 4.1. Créer une table

L'instruction CREATE TABLE de SQLite est utilisée pour créer une nouvelle table dans une base de données. Vous devez définir les noms des colonnes et le type de données de chaque colonne. Voici la syntaxe de base de l'instruction CREATE TABLE :

```
CREATE TABLE nom_bas_de_donnees.nom_table (
   colonne_1 type_donnees,
   colonne_2 type_donnees,
   column3 type_donnees,
   .....
   colonne_n type_donnees,
   PRIMARY KEY (une ou plusieurs colonnes)
);
```

Exemple :
Voici un exemple qui crée une table EMPLOYEE avec ID comme clé primaire et NOT NULL sont les contraintes indiquant que ces champs ne peuvent pas être NULL lors de la création d'enregistrements dans cette table.

```
CREATE TABLE EMPLOYEE (
   ID         INT      PRIMARY KEY,
   NOM        TEXT     NOT NULL,
   AGE        INT      NOT NULL,
   ADRESSE    CHAR(50),
   SALAIRE    REAL
);
```

Créons un autre tableau, que nous utiliserons dans les exercices suivants :

```
CREATE TABLE DEPARTMENT (
   ID           INT              PRIMARY KEY,
   DEPT         CHAR (50)        NOT NULL,
   EMP_ID       INT              NOT NULL
);
```

Exemple de création de la table EMPLOYEE avec un script Python via le module sqlite3 :

```
>>> import sqlite3
>>> connexion = sqlite3.connect ('nom_base_de_donnees.db')
>>> cur = connexion.cursor()
```

```
>>> requete_sql = """
CREATE TABLE EMPLOYEE (
   ID          INT     PRIMARY KEY,
   NOM         TEXT    NOT NULL,
   AGE         INT     NOT NULL,
   ADRESSE     CHAR (50),
   SALAIRE     REAL
) ;
"""
>>> cur.execute(requete_sql)
>>> requete_sql = """
CREATE TABLE DEPARTMENT (
   ID           INT              PRIMARY KEY,
   DEPT         CHAR (50)        NOT NULL,
   EMP_ID       INT              NOT NULL
);
"""
>>> cur.execute(requete_sql)
>>> connexion.commit()
>>> connexion.close()
```

Vous pouvez vérifier, avec l'outil **DB Browser for SQLite**, si les tables ont été créées avec succès.

## 4.2. Les contraintes

Les contraintes sont les règles appliquées à une colonne de données sur une table. Celles-ci permettent de limiter le type de données pouvant enregistrer dans une table. Cela garantit l'exactitude et la fiabilité des données de la base de données.

Les contraintes peuvent être au niveau de la colonne ou de la table. Les contraintes de niveau de colonne ne s'appliquent que sur une seule colonne, alors que les contraintes de niveau de table s'appliquent à l'ensemble de la table.

Voici les contraintes couramment utilisées disponibles dans SQLite :

- La contrainte **NOT NULL :** Garantit qu'une colonne ne peut pas avoir la valeur NULL.
- La contrainte **DEFAULT :** Fournit une valeur par défaut pour une colonne lorsqu'aucune n'est spécifiée.
- La contrainte **UNIQUE :** Garantit que toutes les valeurs d'une colonne sont différentes.
- La contrainte **PRIMARY KEY** : Identifie de manière unique chaque ligne / enregistrement dans une table de base de données.
- La contrainte de vérification **CHECK** : garantit que toutes les valeurs d'une colonne satisfont à certaines conditions.

### 4.2.1. La contrainte NOT NULL

Par défaut, une colonne peut contenir des valeurs NULL. Si vous ne voulez pas qu'une colonne ait une valeur NULL, vous devez définir cette contrainte sur cette colonne en spécifiant que NULL n'est plus autorisé pour cette colonne. Un NULL n'est pas la même chose que *pas de données*, il représente plutôt des données inconnues.

NULL est le terme utilisé pour représenter une valeur manquante. Un champ avec une valeur NULL est un champ **sans valeur** (champ laissé vide lors de la création de l'enregistrement). Il est très important de comprendre qu'une valeur NULL est différente d'une valeur zéro ou d'un champ contenant des espaces.

Exemple :

Par exemple, l'instruction SQLite suivante crée une nouvelle table appelée EMPLOYEE et ajoute cinq colonnes, dont trois, ID, NOM et AGE, spécifient de ne pas accepter les valeurs NULL.

```
CREATE TABLE EMPLOYEE (
   ID          INT      PRIMARY KEY     NOT NULL,
   NOM         TEXT     NOT NULL,
   AGE         INT      NOT NULL,
   ADRESSE     CHAR(50),
   SALAIRE     REAL
);
```

### 4.2.2. La contrainte DEFAULT

La contrainte DEFAULT fournit une valeur par défaut à une colonne lorsque l'instruction INSERT INTO ne fournit pas de valeur spécifique.

Exemple :

Par exemple, l'instruction SQLite suivante crée une nouvelle table appelée EMPLOYEE et ajoute cinq colonnes. Ici, la colonne SALAIRE est définie par défaut sur 5000.00. Par conséquent, si l'instruction INSERT INTO ne fournit pas de valeur pour cette colonne, cette colonne est définie par défaut sur 5000.00.

```
CREATE TABLE EMPLOYEE (
   ID          INT      PRIMARY KEY     NOT NULL,
   NOM         TEXT     NOT NULL,
   AGE         INT      NOT NULL,
   ADRESSE     CHAR(50),
   SALAIRE     REAL     DEFAULT 50000.00
);
```

### 4.2.3. La contrainte UNIQUE

La contrainte UNIQUE empêche que deux enregistrements aient des valeurs identiques dans une colonne particulière. Dans le tableau EMPLOYEE, par exemple, vous pouvez empêcher plusieurs personnes d'avoir le même âge.

Exemple :

Par exemple, l'instruction SQLite suivante crée une nouvelle table appelée EMPLOYEE et ajoute cinq colonnes. Ici, la colonne AGE est définie sur UNIQUE, de sorte que vous ne pouvez pas avoir deux enregistrements du même âge :

```
CREATE TABLE EMPLOYEE (
```

```
   ID          INT      PRIMARY KEY     NOT NULL,
   NOM         TEXT     NOT NULL,
   AGE         INT      NOT NULL  UNIQUE,
   ADRESSE     CHAR(50),
   SALAIRE     REAL     DEFAULT 50000.00
);
```

### 4.2.4. Contrainte de clé principale : PRIMARY KEY

La contrainte PRIMARY KEY identifie de manière unique chaque enregistrement d'une table de base de données. Il peut y avoir plus de colonnes UNIQUE, mais **une seule clé primaire** dans une table. Les clés primaires sont importantes lors de la conception des tables de la base de données. Les clés primaires sont des identifiants uniques.

Nous les utilisons pour faire référence aux lignes de la table. Les clés primaires deviennent des **clés étrangères** dans d'autres tables lors de la création de relations entre les tables.

Remarques :

- Une clé primaire est un champ dans une table qui identifie de manière unique chaque ligne / enregistrement d'une table de base de données.
- Les clés primaires doivent contenir des valeurs uniques.
- Une colonne de clé primaire ne peut pas avoir de valeur NULL.
- **Les clés primaires peuvent être NULL dans SQLite. Ce n'est pas le cas avec d'autres bases de données.**
- Une table ne peut avoir qu'une seule clé primaire, qui peut consister en un ou plusieurs champs. Lorsque plusieurs champs sont utilisés comme clé primaire, ils sont appelés clé composite .
- Si une table a une clé primaire définie sur un ou plusieurs champs, vous ne pouvez pas avoir deux enregistrements ayant la même valeur pour ce ou ces champs.

Exemple :

Vous avez déjà vu plusieurs exemples ci-dessus où nous avons créé une table EMPLOYEE avec ID comme clé primaire.

```
CREATE TABLE EMPLOYEE (
   ID          INT      PRIMARY KEY     NOT NULL,
   NOM         TEXT     NOT NULL,
   AGE         INT      NOT NULL,
   ADRESSE     CHAR(50),
   SALAIRE     REAL
);
```

### 4.2.5. La contrainte de vérification CHECK

La contrainte CHECK permet à une condition de vérifier la valeur saisie dans un enregistrement. Si la condition est évaluée à false, l'enregistrement viole la contrainte et n'est pas entré dans la table.

Exemple :

Par exemple, la requête SQLite suivante crée une nouvelle table appelée EMPLOYEE et ajoute cinq colonnes. Ici, nous ajoutons une colonne CHECK avec SALAIRE, de sorte que vous ne pouvez pas avoir de SALAIRE égale à zéro.

```
CREATE TABLE EMPLOYEE (
   ID          INT      PRIMARY KEY     NOT NULL,
   NOM         TEXT     NOT NULL,
   AGE         INT      NOT NULL,
   ADRESSE     CHAR(50),
   SALAIRE     REAL     CHECK (SALAIRE > 0)
);
```

## 4.3. Modifier la structure d'une table : ALTER TABLE

La commande SQLite **ALTER TABLE** modifie une table existante sans effectuer de vidage ni de rechargement complet des données. Vous pouvez renommer une table à l'aide de l'instruction ALTER TABLE. Des colonnes supplémentaires peuvent être ajoutées à une table existante à l'aide de l'instruction ALTER TABLE.

La commande ALTER TABLE ne prend en charge aucune autre opération dans SQLite, sauf renommer une table et ajouter une colonne à une table existante.

Syntaxe :

Voici la syntaxe de base de ALTER TABLE pour RENOMMER une table existante :

```
ALTER TABLE nom_bd.nom_table RENAME TO nouvel_nome_table;
```

Voici la syntaxe de base de ALTER TABLE pour ajouter une nouvelle colonne dans une table existante :

```
ALTER TABLE nom_bd.nom_table ADD COLUMN nom_colonne...;
```

Exemples :

```
ALTER TABLE EMPLOYEE RENAME TO OLD_EMPLOYEE;
```

L'instruction SQLite ci-dessus renommera la table EMPLOYEE en OLD_ EMPLOYEE.

Maintenant, essayons d'ajouter une nouvelle colonne dans la table OLD_ EMPLOYEE comme suit :

```
ALTER TABLE OLD_EMPLOYEE ADD COLUMN GENRE char(1);
```

Il convient de noter que la colonne nouvellement ajoutée est remplie avec des valeurs NULL.

## 4.4. Supprimer une table

L'instruction SQLite **DROP TABLE** permet de supprimer une définition de table ainsi que toutes les données, contraintes et autorisations associées pour cette table.

Vous devez faire attention en utilisant cette commande car une fois la table supprimée, toutes les informations disponibles dans la table seraient également perdues à jamais.

Voici la syntaxe de base de l'instruction DROP TABLE. Vous pouvez éventuellement spécifier le nom de la base de données avec le nom de la table comme suit :

```
DROP TABLE database_NOM.nom_table;
```

Vérifions d'abord la table EMPLOYEE et ensuite nous la supprimerions de la base de données.

```
DROP TABLE EMPLOYEE;
```

# 5. Manipulation des données

## 5.1. INSERT INTO

L'instruction SQLite **INSERT INTO** est utilisée pour ajouter de nouvelles lignes de données dans une table de la base de données.
Voici les deux syntaxes de base de l'instruction INSERT INTO :

```
INSERT INTO NOM_TABLE [(colonne_1, colonne_2, column3,..., colonne_n)]
VALUES (valeur1, valeur2, valeur3,...,valeurN);
```

Colonne_1, colonne_2, ... colonne_n sont les noms des colonnes de la table dans lesquelles vous souhaitez insérer des données.

Il se peut que vous n'ayez pas besoin de spécifier le nom de la ou des colonnes dans la requête SQLite si vous ajoutez des valeurs pour toutes les colonnes de la table. Cependant, assurez-vous que l'ordre des valeurs est dans le même ordre que celui des colonnes du tableau.

La syntaxe SQLite INSERT INTO serait la suivante :

```
INSERT INTO NOM_TABLE VALUES (valeur1, valeur2, valeur3,...,valeurN);
```

Exemples :

Maintenant, les instructions suivantes créeraient deux enregistrements dans la table EMPLOYEE.

```
INSERT INTO EMPLOYEE (ID,NOM,AGE,ADRESSE,SALAIRE)
VALUES (1, 'Nawel', 32, 'Chebba', 20000.00 );

INSERT INTO EMPLOYEE (ID,NOM,AGE,ADRESSE,SALAIRE)
VALUES (2, 'Ali', 25, 'Tabarka', 15000.00 );

INSERT INTO EMPLOYEE (ID,NOM,AGE,ADRESSE,SALAIRE)
VALUES (3, 'Tarak', 23, 'Nabeul', 20000.00 );

INSERT INTO EMPLOYEE (ID,NOM,AGE,ADRESSE,SALAIRE)
VALUES (4, 'Mariam', 25, 'Raoued ', 65000.00 );

INSERT INTO EMPLOYEE (ID,NOM,AGE,ADRESSE,SALAIRE)
VALUES (5, 'Dalel', 27, 'Tabarka', 85000.00 );

INSERT INTO EMPLOYEE (ID,NOM,AGE,ADRESSE,SALAIRE)
VALUES (6, 'Karim', 22, 'Sousse', 45000.00 );
```

Vous pouvez créer un enregistrement dans la table EMPLOYEE en utilisant la deuxième syntaxe suivante:

```
INSERT INTO EMPLOYEE VALUES (7, 'Jamil', 24, 'Hammamet', 10000.00 );
```

Toutes les déclarations ci-dessus créeraient les enregistrements suivants dans la table EMPLOYEE. Dans la section suivante, vous apprendrez à afficher tous ces enregistrements à partir d'une table.

### 5.1.1. INSERT & PYTHON

Dans un script Python, vous pouvez insérer un enregistrement dans une table en utilisant une requête SQL paramétrée.
Syntaxe :

```
cursor.execute (requete_sql [, paramètres])
```

Exemple :

```
cursor.execute ("INSERT INTO EMPLOYEE VALUES (?,?,?,?,?)",
(7, 'Jamil', 24, 'Hammamet', 10000.00 ))
```

Dans un script Python, vous pouvez insérer plusieurs enregistrements dans la table EMPLOYEE en utilisant la fonction SQLite **executemany** syntaxe suivante :

```
cursor.executemany (requete_sql, liste_de_parameters)
```

Cette méthode exécute une requête SQL en itérant à travers la séquence de paramètres, en passant chaque fois les paramètres en cours à la méthode **execute( )**.

Exemple :

```
cursor.executemany("INSERT INTO EMPLOYEE VALUES (?,?,?,?,?)",
[(1, 'Nawel', 32, 'Chebba', 20000.00 ),
 (2, 'Ali', 25, 'Tabarka', 15000.00 ),
(3, 'Tarak', 23, 'Nabeul', 20000.00 ),
(4, 'Mariam', 25, 'Raoued ', 65000.00 ),
(5, 'Dalel', 27, 'Tabarka', 85000.00 ),
(6, 'Karim', 22, 'Sousse', 45000.00 )
])
```

## 5.2. SELECT

L'instruction SQLite **SELECT** permet d'extraire les données d'une table de base de données SQLite qui renvoie des données sous la forme d'une table de résultats. Ces tables de résultats sont également appelées jeux de résultats.
Voici la syntaxe de base de l'instruction SQLite SELECT.

```
SELECT colonne_1, colonne_2, colonne_n FROM nom_table;
```

Ici, colonne_1, colonne_2 ... sont les champs d'une table dont vous voulez récupérer les valeurs. Si vous voulez récupérer tous les champs disponibles dans le champ, vous pouvez utiliser la syntaxe suivante :

```
SELECT * FROM nom_table;
```

### 5.2.1. SELECT & PYTHON

#### 5.2.1.1. Extraire une ligne du résultat de SELECT

```
cursor.fetchone () :
```

Cette méthode extrait la ligne suivante d'un ensemble de résultats de requête SELECT. Elle renvoie un seul tuple ou None lorsqu'aucune donnée supplémentaire n'est disponible.

Exemple :

```
cursor.execute("SELECT * FROM EMPLOYEE")
employee = cursor.fetchone()
#retourne dans employee un tuple représentant la première ligne de l'ensemble des employees retourné par la requête SQLite SELECT
print(employee[0], employee[1])
```

#### 5.2.1.2. Extraire plusieurs lignes durésultat de SELECT

```
cursor.fetchmany ([size = cursor.arraysize])
```

Cette méthode extrait l'ensemble de lignes d'un résultat de requête SELECT, renvoyant une liste. Une liste vide est renvoyée lorsque plus de lignes sont disponibles. La méthode tente d'extraire autant de lignes que l'indique le paramètre size.

Exemple :

```
cursor.execute("SELECT * FROM EMPLOYEE")
employees = cursor.fetchmany(3)
#retourne dans employees une liste des 3 premières lignes (liste de 3 tuples) de l'ensemble des employees retourné par la requête SQLite SELECT
For e in employees :
    print(e[0],e[1])
```

#### 5.2.1.3. Extraire toutes les lignes du résultat de SELECT

```
cursor.fetchall()
```

Cette méthode récupère toutes les lignes (restantes) d'un résultat de requête SELECT, renvoyant une liste. Une liste vide est renvoyée lorsqu'aucune ligne n'est disponible.

Exemple :

```
cursor.execute("SELECT * FROM EMPLOYEE")
employees = cursor.fetchall()
```

## 5.3. DISTINCT

Le mot clé SQLite **DISTINCT** est utilisé avec l'instruction SELECT pour éliminer tous les enregistrements en double et extraire uniquement les enregistrements uniques.

Il peut arriver que vous ayez plusieurs enregistrements en double dans une table. Lors de l'extraction de tels enregistrements, il est plus judicieux de n'extraire que des enregistrements uniques au lieu d'extraire des enregistrements en double.

Voici la syntaxe de base du mot clé DISTINCT pour éliminer les enregistrements en double.

```
SELECT DISTINCT colonne_1, colonne_2, ..., colonne_n FROM nom_table
```

Exemple :

Considérez la table EMPLOYEE avec les enregistrements suivants :

```
ID          NOM         AGE         ADRESSE             SALAIRE
----------  ----------  ----------  ----------   ----------
1           Nawel       32          Chebba              20000.0
2           Ali         25          Tabarka             15000.0
3           Tarak       23          Nabeul              20000.0
4           Mariam      25          Raoued      65000.0
5           Dalel       27          Tabarka             85000.0
6           Karim       22          Sousse      45000.0
7           Jamil       24          Hammamet            10000.0
8           Nawel       24          Hammamet            20000.0
9           Jamil       44          Nabeul              5000.0
10          Jamil       45          Tabarka             5000.0
```

Voyons d'abord comment la requête SELECT suivante renvoie des enregistrements de salaire en double.

```
SELECT NOM FROM EMPLOYEE;
```

Cela produira le résultat suivant.

```
NOM
----------
Nawel
Ali
Tarak
Mariam
Dalel
Karim
Jamil
Nawel
Jamil
Jamil
```

Utilisons maintenant le mot clé DISTINCT avec la requête SELECT ci-dessus et voyons le résultat.

```
SELECT DISTINCT NOM FROM EMPLOYEE;
```

Cela produira le résultat suivant, où il n'y a aucune entrée dupliquée.

```
NOM
----------
Nawel
Ali
```

```
Tarak
Mariam
Dalel
Karim
Jamil
```

## 5.4. WHERE

La clause SQLite **WHERE** est utilisée pour spécifier une condition lors de l'extraction des données d'une ou de plusieurs tables.

Si la condition donnée est satisfaite, elle renvoie la valeur spécifique de la table. Vous devrez utiliser la clause WHERE pour filtrer les enregistrements et extraire uniquement les enregistrements nécessaires.

La clause WHERE est non seulement utilisée dans l'instruction SELECT, mais également dans les instructions UPDATE, DELETE, etc., qui seront abordées dans les sections suivantes.

Voici la syntaxe de base de l'instruction SQLite SELECT avec la clause WHERE.

```
SELECT colonne_1, colonne_2, colonne_n
FROM   nom_table
WHERE  [condition]
```

Vous pouvez spécifier une condition à l'aide d'opérateurs logiques ou de comparaison tels que >, <, =, LIKE, NOT, etc.

Exemple :

```
ID          NOM         AGE         ADRESSE             SALAIRE
----------  ----------  ----------  ----------          ----------
1           Nawel       32          Chebba      20000.0
2           Ali         25          Tabarka             15000.0
3           Tarak       23          Nabeul              20000.0
4           Mariam      25          Raoued      65000.0
5           Dalel       27          Tabarka             85000.0
6           Karim       22          Sousse      45000.0
7           Jamil       24          Hammamet            10000.0
```

Voici quelques exemples simples illustrant l'utilisation des opérateurs logiques SQLite. L'instruction SELECT suivante répertorie tous les enregistrements pour lesquels AGE est supérieur ou égal à 25 **ET** le salaire est supérieur ou égal à 65000,00.

```
SELECT * FROM EMPLOYEE WHERE AGE >= 25 AND SALAIRE >= 65000;

ID          NOM         AGE          ADRESSE             SALAIRE
----------  ----------  ----------  ----------   ----------
4           Mariam      25          Raoued      65000.0
5           Dalel       27          Tabarka             85000.0
```

L'instruction SELECT suivante répertorie tous les enregistrements pour lesquels AGE est supérieur ou égal à 25 **OU** le salaire est supérieur ou égal à 65000,00.

```
SELECT * FROM EMPLOYEE WHERE AGE >= 25 OR SALAIRE >= 65000;

ID          NOM         AGE         ADRESSE     SALAIRE
----------  ----------  ----------  ----------  ----------
1           Nawel       32          Chebba              20000.0
2           Ali         25          Tabarka     15000.0
4           Mariam      25          Raoued      65000.0
5           Dalel       27          Tabarka             85000.0
```

L'instruction SELECT suivante répertorie tous les enregistrements pour lesquels AGE n'est pas NULL, c'est-à-dire tous les enregistrements car aucun enregistrement n'a AGE égal à NULL.

```
SELECT * FROM EMPLOYEE WHERE AGE IS NOT NULL;

ID          NOM         AGE         ADRESSE     SALAIRE
----------  ----------  ----------  ----------  ----------
1           Nawel       32          Chebba              20000.0
2           Ali         25          Tabarka             15000.0
3           Tarak       23          Nabeul      20000.0
4           Mariam      25          Raoued      65000.0
5           Dalel       27          Tabarka             85000.0
6           Karim       22          Sousse              45000.0
7           Jamil       24          Hammamet            10000.0
```

L'instruction SELECT suivante répertorie tous les enregistrements pour lesquels NOM commence par «Ki», peu importe ce qui vient après «Ki».

```
SELECT * FROM EMPLOYEE WHERE NOM LIKE 'Ki%';

ID          NOM         AGE         ADRESSE     SALAIRE
----------  ----------  ----------  ----------  ----------
6           Karim       22          Sousse              45000.0
```

L'instruction SELECT suivante répertorie tous les enregistrements pour lesquels la valeur AGE est 25 ou 27.

```
SELECT * FROM EMPLOYEE WHERE AGE IN (25, 27);

ID          NOM         AGE         ADRESSE     SALAIRE
----------  ----------  ----------  ----------  ----------
2           Ali         25          Tabarka             15000.0
4           Mariam      25          Raoued      65000.0
5           Dalel       27          Tabarka             85000.0
```

L'instruction SELECT suivante répertorie tous les enregistrements pour lesquels la valeur AGE n'est ni 25 ni 27.

```
SELECT * FROM EMPLOYEE WHERE AGE NOT IN (25, 27);

ID          NOM         AGE         ADRESSE     SALAIRE
```

```
----------  ----------  ----------  ----------  ----------
1           Nawel       32          Chebba              20000.0
3           Tarak       23          Nabeul      20000.0
6           Karim       22          Sousse              45000.0
7           Jamil       24          Hammamet            10000.0
```

L'instruction SELECT suivante répertorie tous les enregistrements dont la valeur AGE est comprise entre 25 et 27.

```
SELECT * FROM EMPLOYEE WHERE AGE BETWEEN 25 AND 27;

ID          NOM         AGE         ADRESSE     SALAIRE
----------  ----------  ----------  ----------  ----------
2           Ali         25          Tabarka             15000.0
4           Mariam      25          Raoued      65000.0
5           Dalel       27          Tabarka             85000.0
```

## 5.5. UPDATE

La requête SQLite **UPDATE** est utilisée pour modifier les enregistrements existants dans une table. Vous pouvez utiliser la clause WHERE avec la requête UPDATE pour mettre à jour les lignes sélectionnées, sinon toutes les lignes seraient mises à jour.

Voici la syntaxe de base d'une requête UPDATE avec la clause WHERE :

```
UPDATE nom_table
SET    colonne_1 = valeur1, colonne_2 = valeur2...., colonne_n = valeurN
WHERE  [condition];
```

Vous pouvez combiner un nombre N de conditions à l'aide d'opérateurs AND ou OR.

Exemple :

Considérez la table EMPLOYEE avec les enregistrements suivants :

```
ID          NOM         AGE         ADRESSE     SALAIRE
----------  ----------  ----------  ----------  ----------
1           Nawel       32          Chebba              20000.0
2           Ali         25          Tabarka             15000.0
3           Tarak       23          Nabeul      20000.0
4           Mariam      25          Raoued      65000.0
5           Dalel       27          Tabarka             85000.0
6           Karim       22          Sousse              45000.0
7           Jamil       24          Hammamet            10000.0
```

Voici un exemple qui mettra à jour ADRESSE pour un client dont l'ID est 6.

```
UPDATE EMPLOYEE SET ADRESSE = 'Tabarka' WHERE ID = 6;
```

Maintenant, la table EMPLOYEE aura les enregistrements suivants.

```
ID          NOM         AGE         ADRESSE     SALAIRE
```

```
----------  ----------  ----------  ----------  ----------
1           Nawel       32          Chebba              20000.0
2           Ali         25          Tabarka             15000.0
3           Tarak       23          Nabeul      20000.0
4           Mariam      25          Raoued      65000.0
5           Dalel       27          Tabarka             85000.0
6           Karim       22          Tabarka             45000.0
7           Jamil       24          Hammamet            10000.0
```

Si vous souhaitez modifier toutes les valeurs de colonne ADRESSE et SALAIRE dans la table EMPLOYEE, vous n'avez pas besoin d'utiliser la clause WHERE et la requête UPDATE sera comme suit :

```
UPDATE EMPLOYEE SET ADRESSE = 'Tabarka', SALAIRE = 20000.00;
```

Maintenant, la table EMPLOYEE aura les enregistrements suivants :

```
ID          NOM         AGE         ADRESSE      SALAIRE
----------  ----------  ----------  ----------   ----------
1           Nawel       32          Tabarka      20000.0
2           Ali         25          Tabarka      20000.0
3           Tarak       23          Tabarka      20000.0
4           Mariam      25          Tabarka      20000.0
5           Dalel       27          Tabarka      20000.0
6           Karim       22          Tabarka      20000.0
7           Jamil       24          Tabarka      20000.0
```

## 5.6. DELETE

La requête SQLite DELETE est utilisée pour supprimer les enregistrements existants d'une table. Vous pouvez utiliser la clause WHERE avec la requête DELETE pour supprimer les lignes sélectionnées, sinon tous les enregistrements seraient supprimés.

Voici la syntaxe de base d'une requête DELETE avec la clause WHERE :

```
DELETE FROM nom_table
WHERE [condition];
```

Vous pouvez combiner un nombre N de conditions à l'aide d'opérateurs AND ou OR.

Exemple :

Considérez la table EMPLOYEE avec les enregistrements suivants.

```
ID          NOM         AGE         ADRESSE     SALAIRE
----------  ----------  ----------  ----------  ----------
1           Nawel       32          Chebba              20000.0
2           Ali         25          Tabarka             15000.0
3           Tarak       23          Nabeul      20000.0
4           Mariam      25          Raoued      65000.0
5           Dalel       27          Tabarka             85000.0
6           Karim       22          Sousse              45000.0
7           Jamil       24          Hammamet    10000.0
```

Voici un exemple qui supprime un client dont l'identifiant est 7.

```
DELETE FROM EMPLOYEE WHERE ID = 7;
```

Maintenant la table EMPLOYEE aura les enregistrements suivants.

```
ID          NOM         AGE         ADRESSE     SALAIRE
----------  ----------  ----------  ----------  ----------
1           Nawel       32          Chebba              20000.0
2           Ali         25          Tabarka             15000.0
3           Tarak       23          Nabeul      20000.0
4           Mariam      25          Raoued      65000.0
5           Dalel       27          Tabarka             85000.0
6           Karim       22          Sousse              45000.0
```

Si vous souhaitez supprimer tous les enregistrements de la table EMPLOYEE, vous n'avez pas besoin d'utiliser la clause WHERE avec la requête DELETE, qui sera comme suit :

```
DELETE FROM EMPLOYEE;
```

Maintenant, la table EMPLOYEE n'a aucun enregistrement car tous les enregistrements ont été supprimés par l'instruction DELETE.

## 5.7. ORDER BY

La clause SQLite **ORDER BY** permet de trier les données dans un ordre croissant ou décroissant, en fonction d'une ou de plusieurs colonnes.

Voici la syntaxe de base de la clause ORDER BY :

```
SELECT liste_colonnes
FROM nom_table
[WHERE condition]
[ORDER BY colonne_1, colonne_2,…, colonne_n] [ASC | DESC];
```

Vous pouvez utiliser plusieurs colonnes dans la clause ORDER BY. Assurez-vous que quelle que soit la colonne que vous utilisez pour trier, cette colonne doit être disponible dans la liste de colonnes choisies devant la clause SELECT.

Exemple :

Considérez la table EMPLOYEE avec les enregistrements suivants :

```
ID          NOM         AGE         ADRESSE     SALAIRE
----------  ----------  ----------  ----------  ----------
1           Nawel       32          Chebba              20000.0
2           Ali         25          Tabarka             15000.0
3           Tarak       23          Nabeul      20000.0
4           Mariam      25          Raoued      65000.0
5           Dalel       27          Tabarka             85000.0
6           Karim       22          Sousse              45000.0
7           Jamil       24          Hammamet            10000.0
```

Voici un exemple qui triera le résultat par ordre décroissant par SALAIRE.

```
SELECT * FROM EMPLOYEE
ORDER BY SALAIRE ASC;
```

Cela produira le résultat suivant.

```
ID          NOM         AGE         ADRESSE     SALAIRE
----------  ----------  ----------  ----------  ----------
7           Jamil       24          Hammamet            10000.0
2           Ali         25          Tabarka             15000.0
1           Nawel       32          Chebba              20000.0
3           Tarak       23          Nabeul      20000.0
6           Karim       22          Sousse              45000.0
4           Mariam      25          Raoued      65000.0
5           Dalel       27          Tabarka             85000.0
```

Voici un exemple, qui triera le résultat par ordre croissant par NOM et SALAIRE.

```
SELECT * FROM EMPLOYEE ORDER BY NOM, SALAIRE ASC;
```

Cela produira le résultat suivant :

```
ID          NOM         AGE         ADRESSE     SALAIRE
----------  ----------  ----------  ----------  ----------
2           Ali         25          Tabarka     15000.0
5           Dalel       27          Tabarka     85000.0
7           Jamil       24          Hammamet    10000.0
6           Karim       22          Sousse      45000.0
4           Mariam      25          Raoued      65000.0
1           Nawel       32          Chebba      20000.0
3           Tarak       23          Nabeul      20000.0
```

Voici un exemple qui triera le résultat par ordre décroissant par NOM.

```
SELECT * FROM EMPLOYEE ORDER BY NOM DESC;
```

Cela produira le résultat suivant.

```
ID          NOM         AGE         ADRESSE       SALAIRE
----------  ----------  ----------  ----------    ----------
3           Tarak       23          Nabeul        20000.0
1           Nawel       32          Chebba        20000.0
4           Mariam      25          Raoued        65000.0
6           Karim       22          Sousse        45000.0
7           Jamil       24          Hammamet      10000.0
5           Dalel       27          Tabarka       85000.0
2           Ali         25          Tabarka       15000.0
```

## 5.8. GROUP BY

La clause SQLite **GROUP BY** est utilisée en collaboration avec l'instruction SELECT pour organiser des données identiques en groupes. La clause GROUP BY suit la clause WHERE dans une instruction SELECT et précède la clause ORDER BY.

Voici la syntaxe de base de la clause GROUP BY :

```
SELECT liste_colonnes
FROM nom_table
WHERE [conditions]
GROUP BY colonne_1, colonne_2, ..., colonne_n
ORDER BY colonne_1, colonne_2, ..., colonne_n
```

Vous pouvez utiliser plusieurs colonnes dans la clause GROUP BY. Assurez-vous que quelle que soit la colonne que vous utilisez pour grouper, cette colonne doit être disponible dans la liste de colonnes.

Exemple :

Considérez la table EMPLOYEE avec les enregistrements suivants :

```
ID          NOM         AGE         ADRESSE     SALAIRE
----------  ----------  ----------  ----------  ----------
1           Nawel       32          Chebba              20000.0
2           Ali         25          Tabarka             15000.0
3           Tarak       23          Nabeul      20000.0
4           Mariam      25          Raoued      65000.0
5           Dalel       27          Tabarka             85000.0
6           Karim       22          Sousse              45000.0
7           Jamil       24          Hammamet            10000.0
```

Si vous souhaitez connaître le montant total du salaire de chaque client, la requête GROUP BY sera la suivante:

```
SELECT NOM, SUM (SALAIRE)
FROM EMPLOYEE
GROUP BY NOM;
```

Cela produira le résultat suivant :

```
NOM            SUM(SALAIRE)
----------     -----------
Ali             15000.0
Dalel           85000.0
Jamil           10000.0
Karim          45000.0
Mariam         65000.0
Nawel           20000.0
Tarak           20000.0
```

Créons maintenant trois autres enregistrements dans la table EMPLOYEE à l'aide des instructions INSERT suivantes.

```
INSERT INTO EMPLOYEE VALUES (8, 'Nawel', 24, 'Hammamet', 20000.00);
INSERT INTO EMPLOYEE VALUES (9, 'Jamil', 44, 'Nabeul', 5000.00);
INSERT INTO EMPLOYEE VALUES (10, 'Jamil', 45, 'Tabarka', 5000.00);
```

Maintenant, notre table a les enregistrements suivants avec des noms en double.

```
ID          NOM         AGE         ADRESSE     SALAIRE
```

```
----------  ----------  ----------  ----------  ----------
1           Nawel       32          Chebba                20000.0
2           Ali         25          Tabarka               15000.0
3           Tarak       23          Nabeul      20000.0
4           Mariam      25          Raoued      65000.0
5           Dalel       27          Tabarka               85000.0
6           Karim       22          Sousse                45000.0
7           Jamil       24          Hammamet              10000.0
8           Nawel       24          Hammamet              20000.0
9           Jamil       44          Nabeul      5000.0
10          Jamil       45          Tabarka               5000.0
```

Encore une fois, utilisons la même instruction pour regrouper tous les enregistrements en utilisant la colonne NOM comme suit:

```
SELECT NOM, SUM (SALAIRE)
FROM EMPLOYEE
GROUP BY NOM
ORDER BY NOM;
```

Cela produira le résultat suivant.

```
NOM            SUM (SALAIRE)
----------     -----------
Ali             15000
Dalel           85000
Jamil           20000
Karim          45000
Mariam         65000
Nawel           40000
Tarak           20000
```

## 5.9. HAVING

La clause **HAVING** vous permet de spécifier les conditions permettant de filtrer les résultats du groupe qui apparaissent dans les résultats finaux.

*NB : La clause WHERE place des conditions sur les colonnes sélectionnées, tandis que la clause HAVING place des conditions sur les groupes créés par la clause GROUP BY.*

Voici la position de la clause HAVING dans une requête SELECT :

```
SELECT
FROM
WHERE
GROUP BY
HAVING
ORDER BY
```

La clause HAVING doit suivre la clause GROUP BY dans une requête et doit également précéder la clause ORDER BY si elle est utilisée.

Voici la syntaxe de l'instruction SELECT, y compris la clause HAVING.

```
SELECT colonne_1, colonne_2, …, colonne_n
FROM           table1, table2
WHERE          [conditions]
GROUP BY       colonne_1, colonne_2
HAVING [conditions]
ORDER BY       colonne_1
```

Exemple :

Considérez la table EMPLOYEE avec les enregistrements suivants :

```
ID          NOM         AGE         ADRESSE     SALAIRE
----------  ----------  ----------  ----------  ----------
1           Nawel       32          Chebba              20000.0
2           Ali         25          Tabarka             15000.0
3           Tarak       23          Nabeul      20000.0
4           Mariam      25          Raoued      65000.0
5           Dalel       27          Tabarka             85000.0
6           Karim       22          Sousse              45000.0
7           Jamil       24          Hammamet            10000.0
8           Nawel       24          Hammamet            20000.0
9           Jamil       44          Nabeul      5000.0
10          Jamil       45          Tabarka             5000.0
```

Voici l'exemple qui affichera l'enregistrement pour lequel le nombre de noms est inférieur à 2.

```
SELECT *
FROM EMPLOYEE
GROUP BY NOM
HAVING count (NOM) < 2;
```

Cela produira le résultat suivant :

```
ID          NOM         AGE         ADRESSE     SALAIRE
----------  ----------  ----------  ----------  ----------
2           Ali         25          Tabarka     15000
5           Dalel       27          Tabarka     85000
6           Karim       22          Sousse           45000
4           Mariam      25          Raoued      65000
3           Tarak       23          Nabeul      20000
```

## 5.10. LIMIT

La clause SQLite **LIMIT** est utilisée pour limiter le nombre des enregistrements renvoyés par l'instruction SELECT.

Voici la syntaxe de base de l'instruction SELECT avec la clause LIMIT :

```
SELECT colonne_1, colonne_2, …, colonne_n
FROM           nom_table
LIMIT          [nombre_de_lignes]
```

Voici la syntaxe de la clause LIMIT lorsqu'elle est utilisée avec la clause OFFSET.

```
SELECT colonne_1, colonne_2, …,colonne_n
FROM           nom_table
LIMIT          [nombre_de_lignes] OFFSET [numero_de_ligne]
```

Le moteur SQLite renverra les lignes à partir de la ligne suivante vers le décalage spécifié, comme indiqué ci-dessous dans le dernier exemple.

Exemple :

Considérez la table EMPLOYEE avec les enregistrements suivants :

```
ID          NOM         AGE         ADRESSE     SALAIRE
----------  ----------  ----------  ----------  ----------
1           Nawel       32          Chebba              20000.0
2           Ali         25          Tabarka             15000.0
3           Tarak       23          Nabeul      20000.0
4           Mariam      25          Raoued      65000.0
5           Dalel       27          Tabarka             85000.0
6           Karim       22          Sousse              45000.0
7           Jamil       24          Hammamet            10000.0
```

Exemple :

```
SELECT * FROM EMPLOYEE LIMIT 6;
```

Cela produira le résultat suivant :

```
ID          NOM         AGE         ADRESSE     SALAIRE
----------  ----------  ----------  ----------  ----------
1           Nawel       32          Chebba              20000.0
2           Ali         25          Tabarka             15000.0
3           Tarak       23          Nabeul      20000.0
4           Mariam      25          Raoued      65000.0
5           Dalel       27          Tabarka             85000.0
6           Karim       22          Sousse              45000.0
```

Cependant, dans certaines situations, vous devrez peut-être récupérer un ensemble d'enregistrements à partir d'un décalage particulier. Voici un exemple, qui ramasse 3 enregistrements à partir de la 3ème position.

```
SELECT * FROM EMPLOYEE LIMIT 3 OFFSET 2;
```

Cela produira le résultat suivant :

```
ID          NOM         AGE         ADRESSE     SALAIRE
----------  ----------  ----------  ----------  ----------
3           Tarak       23          Nabeul      20000.0
4           Mariam      25          Raoued      65000.0
5           Dalel       27          Tabarka     85000.0
```

## 5.11. LIKE

L' opérateur SQLite **LIKE** permet de faire correspondre les valeurs de texte à un modèle à l'aide de caractères génériques. Si l'expression de recherche peut correspondre à l'expression de modèle, l'opérateur LIKE renverra true. Deux caractères génériques sont utilisés conjointement avec l'opérateur LIKE :

- Le signe de pourcentage (**%**) représente **zéro**, **un** ou **plusieurs** chiffres ou caractères.
- Le trait de soulignement ( **_** ) représente **un** chiffre ou **un** caractère unique.

Ces symboles peuvent être utilisés en combinaison.

Syntaxe : Voici la syntaxe de base de **%** et **_**

```
SELECT FROM nom_table
WHERE column LIKE 'XXXX%'
```

```
SELECT FROM nom_table
WHERE column LIKE '%XXXX%'
```

```
SELECT FROM nom_table
WHERE column LIKE 'XXXX_'
```

```
SELECT FROM nom_table
WHERE column LIKE '_XXXX'
```

```
SELECT FROM nom_table
WHERE column LIKE '_XXXX_'
```

Vous pouvez combiner un nombre N de conditions à l'aide d'opérateurs AND ou OR. Ici, XXXX peut être n'importe quelle valeur numérique ou chaîne.

Exemples :

Le tableau suivant répertorie un certain nombre d'exemples montrant une partie WHERE ayant différentes clauses LIKE avec les opérateurs '%' et '_'.

| Déclaration | Description |
|---|---|
| WHERE salaire **LIKE '200%'** | Trouve toutes les valeurs commençant par 200 |
| WHERE salaire **LIKE'%200%'** | Trouve toutes les valeurs qui ont 200 dans n'importe quelle position |
| WHERE salaire **LIKE '_00%'** | Trouve toutes les valeurs qui ont 00 dans les deuxième |

| | |
|---|---|
| | et troisième positions |
| WHERE salaire **LIKE '2 _% _%'** | Recherche les valeurs commençant par 2 et comportant au moins 3 caractères. |
| WHERE salaire **LIKE '%2'** | Trouve les valeurs qui finissent par 2 |
| WHERE salaire **LIKE '_2%3'** | Trouve toutes les valeurs ayant un 2 en deuxième position et se terminant par 3 |
| WHERE salaire **LIKE '2___3'** | Trouve toutes les valeurs dans un nombre à cinq chiffres commençant par 2 et se terminant par 3 |

Prenons un exemple concret, considérons la table EMPLOYEE avec les enregistrements suivants :

```
ID          NOM         AGE         ADRESSE     SALAIRE
----------  ----------  ----------  ----------  ----------
1           Nawel       32          Chebba              20000.0
2           Ali         25          Tabarka             15000.0
3           Tarak       23          Nabeul      20000.0
4           Mariam      25          Raoued      65000.0
5           Dalel       27          Tabarka             85000.0
6           Karim       22          Sousse              45000.0
7           Jamil       24          Hammamet            10000.0
```

Voici un exemple qui affichera tous les enregistrements de la table EMPLOYEE où AGE commence par 2 :

```
SELECT * FROM EMPLOYEE WHERE AGE LIKE '2%';
```

Cela produira le résultat suivant :

```
ID          NOM         AGE         ADRESSE     SALAIRE
----------  ----------  ----------  ----------  ----------
2           Ali         25          Tabarka     15000.0
3           Tarak       23          Nabeul      20000.0
4           Mariam      25          Raoued      65000.0
5           Dalel       27          Tabarka     85000.0
6           Karim       22          Sousse      45000.0
7           Jamil       24          Hammamet    10000.0
```

Voici un exemple qui affichera tous les enregistrements de la table EMPLOYEE où ADRESSE aura un trait d'union (-) à l'intérieur du texte :

```
SELECT * FROM EMPLOYEE WHERE ADRESSE LIKE '%-%';
```

Cela produira le résultat suivant :

```
ID          NOM         AGE         ADRESSE     SALAIRE
----------  ----------  ----------  ----------  ----------
4           Mariam      25          Raoued      65000.0
```

```
6           Karim       22          Sousse      45000.0
```

## 5.12. JOIN

La clause SQLite **JOIN** est utilisée pour combiner les enregistrements de deux tables ou plus dans une base de données. Une jointure est un moyen de combiner des champs de deux tables en utilisant des valeurs communes à chacune.

Avant de commencer, considérons deux tables EMPLOYEE et DEPARTMENT. Nous avons déjà vu des instructions INSERT pour remplir la table EMPLOYEE. Supposons donc que la liste des enregistrements disponibles dans la table EMPLOYEE :

```
ID          NOM         AGE         ADRESSE     SALAIRE
----------  ----------  ----------  ----------  ----------
1           Nawel       32          Chebba              20000.0
2           Ali         25          Tabarka             15000.0
3           Tarak       23          Nabeul      20000.0
4           Mariam      25          Raoued      65000.0
5           Dalel       27          Tabarka             85000.0
6           Karim       22          Sousse              45000.0
7           Jamil       24          Hammamet            10000.0
```

Une autre table est DEPARTEMENT avec la définition suivante :

```
CREATE TABLE DEPARTMENT(
   ID          INT             PRIMARY KEY      NOT NULL,
   DEPTCHAR(50)        NOT NULL,
   EMP_ID      INT             NOT NULL
);
```

Voici la liste des instructions INSERT à remplir pour la table DEPARTMENT :

```
INSERT INTO DEPARTMENT (ID, DEPT, EMP_ID)
VALUES (1, 'Facturation', 1);

INSERT INTO DEPARTMENT (ID, DEPT, EMP_ID)
VALUES (2, 'Ingénierie', 2);

INSERT INTO DEPARTMENT (ID, DEPT, EMP_ID)
VALUES (3, 'Finance', 7);
```

Enfin, nous avons la liste suivante des enregistrements disponibles dans la table DEPARTMENT :

```
ID          DEPT         EMP_ID
----------  ----------   ----------
1           Facturation  1
2           Ingénierie   2
3           Finance      7
```

La clause **JOIN** établit une correspondance entre chaque ligne du premier tableau et chaque ligne du deuxième tableau. Si les tables d'entrée ont respectivement une rangée x et y, la table résultante aura une rangée x * y. Comme la jointure peut générer des tables extrêmement volumineuses, vous devez veiller à ne les utiliser que lorsque cela est approprié.

Voici la syntaxe de JOIN :

```
SELECT ... FROM table1 JOIN table2 ON condition
```

Sur la base des tableaux ci-dessus, vous pouvez écrire un JOIN comme suit :

```
SELECT EMP_ID, NOM, DEPT

FROM EMPLOYEE JOIN DEPARTMENT

ON EMPLOYEE.ID = DEPARTMENT.EMP_ID;
```

La requête ci-dessus produira le résultat suivant :

```
EMP_ID      NOM         DEPT
----------  ----------          ----------
1           Nawel               Facturation
2           Nawel               Ingénierie
7           Nawel               Finance
1           Ali         Facturation
2           Ali         Ingénierie
7           Ali         Finance
1           Tarak               Facturation
2           Tarak               Ingénierie
7           Tarak               Finance
1           Mariam              Facturation
2           Mariam              Ingénierie
7           Mariam              Finance
1           Dalel               Facturation
2           Dalel               Ingénierie
7           Dalel               Finance
1           Karim               Facturation
2           Karim               Ingénierie
7           Karim               Finance
1           Jamil               Facturation
2           Jamil               Ingénierie
7           Jamil               Finance
```

## 5.13. UNION

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

Pour utiliser UNION, chaque SELECT doit avoir le même nombre de colonnes sélectionnées, le même type de données et les avoir dans le même ordre, mais elles ne doivent pas nécessairement être de la même longueur.

Voici la syntaxe de base de UNION .

```
SELECT colonne_1 [, colonne_2]
FROM   table1 [, table2]
[WHERE condition]

UNION

SELECT colonne_1 [, colonne_2]
FROM   table1 [, table2]
[WHERE condition]
```

Ici, la condition donnée peut être une expression donnée en fonction de vos besoins.

Exemple :

Considérez les deux tableaux suivants:

Le tableau EMPLOYEE comme suit :

```
select * from EMPLOYEE;
ID          NOM                 AGE         ADRESSE     SALAIRE
----------  ------------------  ----------  ----------  ----------
1           Nawel               32          Chebba      20000.0
2           Ali                 25          Tabarka     15000.0
3           Tarak               23          Nabeul      20000.0
4           Mariam              25          Raoued      65000.0
5           Dalel               27          Tabarka     85000.0
6           Karim               22          Sousse      45000.0
7           Jamil               24          Hammamet    10000.0
```

Un autre tableau est DÉPARTEMENT comme suit :

```
ID          DEPT                EMP_ID
----------  ------------------  ----------
1           Facturation         1
2           Ingénierie          2
3           Finance             7
4           Ingénierie          3
5           Finance             4
6           Ingénierie          5
7           Finance             6
```

Joignons maintenant ces tables en utilisant l'instruction SELECT avec la clause UNION comme suit :

```
SELECT EMP_ID, NOM, DEPT FROM EMPLOYEE INNER JOIN DEPARTMENT
ON EMPLOYEE.ID = DEPARTMENT.EMP_ID

UNION

SELECT EMP_ID, NOM, DEPT FROM EMPLOYEE LEFT OUTER JOIN DEPARTMENT
```

```
ON EMPLOYEE.ID = DEPARTMENT.EMP_ID;
```

Cela produira le résultat suivant :

```
EMP_ID      NOM                  DEPT
----------  -------------------  ----------
1           Nawel                Facturation
2           Ali                  Ingénierie
3           Tarak                Ingénierie
4           Mariam               Finance
5           Dalel                Ingénierie
6           Karim                Finance
7           Jamil                Finance
```

## 5.14. UNION ALL

L'opérateur UNION ALL permet de combiner les résultats de deux instructions SELECT, y compris les lignes en double.

Les mêmes règles qui s'appliquent à UNION s'appliquent également à l'opérateur UNION ALL.

Voici la syntaxe de base de UNION ALL :

```
SELECT colonne_1 [, colonne_2]
FROM table1 [, table2]
[WHERE condition]

UNION ALL

SELECT colonne_1 [, colonne_2]
FROM table1 [, table2]
[WHERE condition]
```

Ici, la condition donnée peut être une expression donnée en fonction de vos besoins.

Exemple :

Maintenant, rejoignons les deux tables dans notre instruction SELECT comme suit:

```
SELECT EMP_ID, NOM, DEPT FROM EMPLOYEE INNER JOIN DEPARTMENT

ON EMPLOYEE.ID = DEPARTMENT.EMP_ID

UNION ALL

SELECT EMP_ID, NOM, DEPT FROM EMPLOYEE LEFT OUTER JOIN DEPARTMENT

ON EMPLOYEE.ID = DEPARTMENT.EMP_ID;
```

Cela produira le résultat suivant :

```
EMP_ID      NOM                  DEPT
----------  -------------------  ----------
1           Nawel                Facturation
2           Ali                  Ingénierie
3           Tarak                Ingénierie
```

```
4           Mariam              Finance
5           Dalel               Ingénierie
6           Karim               Finance
7           Jamil               Finance
1           Nawel               Facturation
2           Ali                 Ingénierie
3           Tarak               Ingénierie
4           Mariam              Finance
5           Dalel               Ingénierie
6           Karim               Finance
7           Jamil               Finance
```

# 6. SQLite : propriétés

## 6.1. Types de données

Le type de données SQLite est un attribut qui spécifie le type de données d'un objet. Chaque colonne, variable et expression a un type de données associé dans SQLite.

Le tableau suivant répertorie divers noms de types de données pouvant être utilisés lors de la création de tables SQLite3 :

| Type de données générique | Ttypes de données affinés |
|---|---|
| INTEGER | INT, INTEGER, TINYINT, SMALLINT, … |
| TEXT | VARCHAR(255), TEXT, … |
| REAL | REAL, DOUBLE, FLOAT, … |
| NUMERIC | DECIMAL(10,5), BOOLEAN, DATE, DATETIME, … |

Remarques :

- Les valeurs booléennes sont stockées en tant que nombres entiers 0 (faux) et 1 (vrai).
- SQLite est capable de stocker les dates et les heures sous forme de valeurs de type TEXT. Exemple d'une date dans un format du type TEXT "AAAA-MM-JJ HH: MM: SS.SSS"

## 6.2. NULL

NULL est le terme utilisé pour représenter une valeur manquante. Une valeur NULL dans une table est une valeur dans un champ qui semble être vide.

Un champ avec une valeur NULL est un champ **sans valeur** (champ laissé vide lors de la création de l'enregistrement). Il est très important de comprendre qu'une valeur NULL est différente d'une valeur zéro ou d'un champ contenant des espaces.

Syntaxe :

Voici la syntaxe de base de l'utilisation de NULL lors de la création d'une table :

```
CREATE TABLE EMPLOYEE (
   ID          INT      PRIMARY KEY     NOT NULL,
   NOM         TEXT     NOT NULL,
   AGE         INT      NOT NULL,
   ADRESSE     CHAR(50),
   SALAIRE     REAL
);
```

Ici, NOT NULL signifie que la colonne doit toujours accepter une valeur explicite du type de données. Nous n'avons pas utilisé NOT NULL dans deux colonnes, ce qui signifie que ces colonnes pourraient être NULL.

Exemple :

La valeur NULL peut poser des problèmes lors de la sélection de données, car lors de la comparaison d'une valeur inconnue avec une autre valeur, le résultat est toujours inconnu et n'est pas inclus dans les résultats finaux. Considérez le tableau suivant, `EMPLOYEE` avec les enregistrements suivants :

```
ID          NOM         AGE         ADRESSE     SALAIRE
----------  ----------  ----------  ----------  ----------
1           Nawel       32          Chebba      20000.0
2           Ali         25          Tabarka     15000.0
3           Tarak       23          Nabeul      20000.0
4           Mariam      25          Raoued      65000.0
5           Dalel       27          Tabarka     85000.0
6           Karim       22          Sousse      45000.0
7           Jamil       24          Hammamet    10000.0
```

Utilisons l'instruction UPDATE pour définir quelques valeurs comme NULL comme suit :

```
UPDATE EMPLOYEE SET ADRESSE = NULL, SALAIRE = NULL where ID IN (6,7);
```

Maintenant, la table EMPLOYEE aura les enregistrements suivants :

```
ID          NOM         AGE         ADRESSE     SALAIRE
----------  ----------  ----------  ----------  ----------
1           Nawel       32          Chebba      20000.0
2           Ali         25          Tabarka     15000.0
3           Tarak       23          Nabeul      20000.0
4           Mariam      25          Raoued      65000.0
5           Dalel       27          Tabarka     85000.0
6           Karim       22
7           Jamil       24
```

Voyons ensuite l'utilisation de l' opérateur IS NOT NULL pour répertorier tous les enregistrements pour lesquels SALAIRE n'est pas NULL.

```
SELECT ID, NOM, AGE, ADRESSE, SALAIRE
FROM   EMPLOYEE
WHERE  SALAIRE IS NOT NULL;
```

L'instruction SQLite ci-dessus produira le résultat suivant:

```
ID          NOM         AGE         ADRESSE     SALAIRE
----------  ----------  ----------  ----------  ----------
1           Nawel       32          Chebba      20000.0
2           Ali         25          Tabarka     15000.0
3           Tarak       23          Nabeul      20000.0
4           Mariam      25          Raoued      65000.0
5           Dalel       27          Tabarka     85000.0
```

Voici l'utilisation de l' opérateur IS NULL , qui répertorie tous les enregistrements pour lesquels SALAIRE est NULL.

```
SELECT    ID, NOM, AGE, ADRESSE, SALAIRE
FROM      EMPLOYEE
WHERE     SALAIRE IS NULL;
```

L'instruction SQLite ci-dessus produira le résultat suivant.

```
ID          NOM         AGE         ADRESSE     SALAIRE
----------  ----------  ----------  ----------  ----------
6           Karim       22
7           Jamil       24
```

## 6.3. AUTOINCREMENT

La clause SQLite **AUTOINCREMENT** est un mot clé utilisé pour incrémenter automatiquement la valeur d'un champ de la table. Nous pouvons auto-incrémenter une valeur de champ en utilisant le mot-clé AUTOINCREMENT lors de la création d'une table avec un nom de colonne spécifique à incrémenter automatiquement.

Remarque : Le mot-clé **AUTOINCREMENT** peut être utilisé avec le champ **INTEGER uniquement**.

Syntaxe :

L'utilisation de base du mot clé AUTOINCREMENT est la suivante :

```
CREATE TABLE nom_table (
   Colonne_1   INTEGER AUTOINCREMENT,
   Colonne_2   type_donnees,
   Column3     type_donnees,
   .....
   Colonne_n   type_donnees,
);
```

Exemple :

Envisagez de créer la table EMPLOYEE comme suit :

```
CREATE TABLE EMPLOYEE (
   ID          INTEGER PRIMARY KEY AUTOINCREMENT,
   NOM TEXT    NOT NULL,
   AGE         INT     NOT NULL,
   ADRESSE     CHAR (50),
   SALAIRE     REAL
);
```

Maintenant, insérez les enregistrements suivants dans la table EMPLOYEE :

```
INSERT INTO EMPLOYEE (NOM, AGE, ADRESSE, SALAIRE)
VALUES ('Nawel', 32, 'Chebba', 20000.00);

INSERT INTO EMPLOYEE (NOM, AGE, ADRESSE, SALAIRE)
VALUES ('Ali', 25, 'Tabarka', 15000.00);

INSERT INTO EMPLOYEE (NOM, AGE, ADRESSE, SALAIRE)
VALUES ('Tarak', 23, 'Nabeul', 20000.00);

INSERT INTO EMPLOYEE (NOM, AGE, ADRESSE, SALAIRE)
VALUES ('Mariam', 25, 'Raoued ', 65000.00);

INSERT INTO EMPLOYEE (NOM, AGE, ADRESSE, SALAIRE)
VALUES ('Dalel', 27, 'Tabarka', 85000.00);

INSERT INTO EMPLOYEE (NOM, AGE, ADRESSE, SALAIRE)
VALUES ('Karim', 22, 'Sousse', 45000.00);

INSERT INTO EMPLOYEE (NOM, AGE, ADRESSE, SALAIRE)
```

```
VALUES ('Jamil', 24, 'Hammamet', 10000.00);
```

Ceci insérera 7 tuples dans le tableau EMPLOYEE :

```
ID          NOM         AGE         ADRESSE     SALAIRE
----------  ----------  ----------  ----------  ----------
1           Nawel       32          Chebba      20000.0
2           Ali         25          Tabarka     15000.0
3           Tarak       23          Nabeul      20000.0
4           Mariam      25          Raoued      65000.0
5           Dalel       27          Tabarka     85000.0
6           Karim       22          Sousse      45000.0
7           Jamil       24          Hammamet    10000.0
```

## 6.4. ALIAS

Vous pouvez renommer temporairement une table ou une colonne en attribuant un autre nom, appelé **ALIAS** . L'utilisation d'alias de table signifie renommer une table dans une instruction SQLite particulière. Renommer est une modification temporaire et le nom de la table réelle ne change pas dans la base de données.

Les alias de colonnes permettent de renommer les colonnes d'une table aux fins d'une requête SQLite particulière.

Voici la syntaxe de base de l' alias de table :

```
SELECT colonne_1, colonne_2....
FROM   nom_table AS alias_NOM
WHERE  [condition];
```

Voici la syntaxe de base de l' alias de colonne :

```
SELECT nom_colonne AS alias_NOM
FROM   nom_table
WHERE  [condition];
```

Exemple :

Considérez les deux tables suivantes:

La table EMPLOYEE est la suivante :

```
select * from EMPLOYEE;

ID          NOM             AGE         ADRESSE    SALAIRE
----------  ------------        ----------  ---------  ----------
1           Nawel           32          Chebba          20000.0
2           Ali             25          Tabarka    15000.0
3           Tarak           23          Nabeul     20000.0
4           Mariam          25          Raoued          65000.0
```

```
5           Dalel           27          Tabarka    85000.0
6           Karim           22          Sousse          45000.0
7           Jamil           24          Hammamet   10000.0
```

Un autre tableau DÉPARTEMENT est comme suit :

```
ID          DEPT                    EMP_ID
----------  --------------------    ----------
1           Facturation             1
2           Ingénierie              2
3           Finance                 7
4           Ingénierie              3
5           Finance                 4
6           Ingénierie              5
7           Finance                 6
```

Maintenant, voici l'utilisation d'alias de table où nous utilisons, respectivement, C et D comme alias pour les tables EMPLOYEE et DEPARTMENT :

```
SELECT C.ID, C.NOM, C.AGE, D.DEPT
FROM   EMPLOYEE AS C, DEPARTMENT AS D
WHERE  C.ID = D.EMP_ID;
```

L'instruction SQLite ci-dessus produira le résultat suivant:

```
ID          NOM         AGE         DEPT
----------  ----------  ----------  ----------
1           Nawel       32          Facturation
2           Ali         25          Ingénierie
3           Tarak       23          Ingénierie
4           Mariam      25          Finance
5           Dalel       27          Ingénierie
6           Karim       22          Finance
7           Jamil       24          Finance
```

Prenons un exemple d'utilisation d'alias de colonne où EMPLOYEE_ID est un alias de colonne ID et EMPLOYEE_NOM est un alias de colonne NOM :

```
SELECT C.ID AS EMPLOYEE_ID, C.NOM AS EMPLOYEE_NOM, C.AGE, D.DEPT
FROM   EMPLOYEE AS C, DEPARTMENT AS D
WHERE  C.ID = D.EMP_ID;
```

L'instruction SQLite ci-dessus produira le résultat suivant:

```
EMPLOYEE_ID             EMPLOYEE_NOM            AGE         DEPT
----------              ------------            ----------  ----------
1                       Nawel                   32          Facturation
2                       Ali                     25          Ingénierie
3                       Tarak                   23          Ingénierie
4                       Mariam          25          Finance
5                       Dalel                   27          Ingénierie
6                       Karim           22          Finance
7                       Jamil                   24          Finance
```

## 7. Sous-requête

Une **sous-requête**, une **requête interne** ou une **requête imbriquée** est une requête dans une autre requête SQLite et incorporée dans la clause WHERE.

Une sous-requête est utilisée pour renvoyer des données qui seront utilisées dans la requête principale comme condition pour limiter davantage les données à récupérer.

Les sous-requêtes peuvent être utilisées avec les instructions SELECT, INSERT, UPDATE et DELETE avec les opérateurs tels que =, <,>,> =, <=, IN, BETWEEN, etc.

Les sous-requêtes doivent respecter quelques règles :

- Les sous-requêtes doivent être entre parenthèses.
- Une sous-requête ne peut contenir qu'une colonne dans la clause SELECT, à moins que plusieurs colonnes ne soient dans la requête principale pour que la sous-requête compare les colonnes sélectionnées.
- Un ORDER BY ne peut pas être utilisé dans une sous-requête, bien que la requête principale puisse utiliser un ORDER BY.
- GROUP BY peut être utilisé pour exécuter la même fonction que ORDER BY dans une sous-requête.
- Les sous-requêtes qui renvoient plusieurs lignes ne peuvent être utilisées qu'avec plusieurs opérateurs de valeur, tels que l'opérateur IN.
- L'opérateur BETWEEN ne peut pas être utilisé avec une sous-requête; Cependant, BETWEEN peut être utilisé dans la sous-requête.

Exemples :

### 7.1. Sous-requêtes avec instruction SELECT

Les sous-requêtes sont le plus souvent utilisées avec l'instruction SELECT.
La syntaxe de base est la suivante :

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

Exemple :
Considérez la table EMPLOYEE avec les enregistrements suivants :

```
ID          NOM         AGE         ADRESSE      SALAIRE
----------  ----------  ----------  ----------   ----------
1           Nawel       32          Chebba               20000.0
2           Ali         25          Tabarka              15000.0
3           Tarak       23          Nabeul       20000.0
4           Mariam      25          Raoued       65000.0
5           Dalel       27          Tabarka              85000.0
6           Karim       22          Sousse               45000.0
```

```
7           Jamil       24          Hammamet            10000.0
```

Vérifions maintenant la sous-requête suivante avec l'instruction SELECT.

```
SELECT *  FROM EMPLOYEE
WHERE ID IN (SELECT ID  FROM EMPLOYEE WHERE SALAIRE > 45000) ;
```

Cela produira le résultat suivant :

```
ID          NOM         AGE         ADRESSE     SALAIRE
----------  ----------  ----------  ----------  ----------
4           Mariam      25          Raoued      65000.0
5           Dalel       27          Tabarka     85000.0
```

## 7.2. Sous-requêtes avec instruction INSERT

Les sous-requêtes peuvent également être utilisées avec les instructions INSERT. L'instruction INSERT utilise les données renvoyées par la sous-requête pour les insérer dans une autre table. Les données sélectionnées dans la sous-requête peuvent être modifiées avec l'une des fonctions de caractère, date ou numéro.

Voici la syntaxe de base est la suivante -

```
INSERT INTO nom_table [(colonne_1 [, colonne_2])]
SELECT *|colonne_1 [, colonne_2]
FROM table1 [, table2]
[WHERE]
```

Exemple :
Considérons une table EMPLOYEE_BKP avec une structure similaire à la table EMPLOYEE et peut être créée à l'aide du même CREATE TABLE utilisant EMPLOYEE_BKP comme nom de table. Pour copier la table COMPTE complète dans EMPLOYEE_BKP, voici la syntaxe :

```
INSERT INTO EMPLOYEE_BKP
SELECT * FROM EMPLOYEE
WHERE ID IN (SELECT ID FROM EMPLOYEE);
```

## 7.3. Sous-requêtes avec instruction UPDATE

La sous-requête peut être utilisée conjointement avec l'instruction UPDATE. Une ou plusieurs colonnes d'une table peuvent être mises à jour lors de l'utilisation d'une sous-requête avec l'instruction UPDATE.

Voici la syntaxe de base est la suivante :

```
UPDATE table
SET nom_colonne = nouvelle_valeur
[WHERE OPERATEUR (SELECT NOM_COLONNE FROM NOM_TABLE [WHERE])]
```

Exemple :

En supposant que nous ayons une table EMPLOYEE_BKP disponible qui est une sauvegarde de la table EMPLOYEE.
L'exemple suivant met à jour SALAIRE de 0,50 fois dans la table EMPLOYEE pour tous les clients dont l'ÂGE est supérieur ou égal à 27.

```
UPDATE EMPLOYEE
SET SALAIRE = SALAIRE * 0.50
WHERE AGE IN (SELECT AGE FROM EMPLOYEE_BKP WHERE AGE >= 27 );
```

Cela aurait un impact sur deux lignes et finalement, la table EMPLOYEE aurait les enregistrements suivants :

```
ID          NOM         AGE         ADRESSE     SALAIRE
----------  ----------  ----------  ----------  ----------
1           Nawel       32          Chebba              10000.0
2           Ali         25          Tabarka             15000.0
3           Tarak       23          Nabeul      20000.0
4           Mariam      25          Raoued      65000.0
5           Dalel       27          Tabarka             42500.0
6           Karim       22          Sousse              45000.0
7           Jamil       24          Hammamet            10000.0
```

## 7.4. Sous-requêtes avec instruction DELETE

La sous-requête peut être utilisée conjointement avec l'instruction DELETE, comme avec toutes les autres instructions mentionnées ci-dessus.

Voici la syntaxe de base est la suivante :

```
DELETE FROM NOM_TABLE
[WHERE OPERATEUR (SELECT NOM_COLONNE FROM NOM_TABLE [WHERE])]
```

Exemple : En supposant que nous ayons une table EMPLOYEE_BKP disponible qui est une sauvegarde de la table EMPLOYEE.

L'exemple suivant supprime les enregistrements de la table EMPLOYEE pour tous les clients dont AGE est supérieur ou égal à 27.

```
DELETE FROM EMPLOYEE
WHERE AGE IN (SELECT AGE FROM EMPLOYEE_BKP WHERE AGE > 27);
```

Cela aura un impact sur deux lignes et enfin la table EMPLOYEE aura les enregistrements suivants :

```
ID          NOM         AGE         ADRESSE     SALAIRE
----------  ----------  ----------  ----------  ----------
2           Ali         25          Tabarka     15000.0
3           Tarak       23          Nabeul      20000.0
4           Mariam      25          Raoued      65000.0
5           Dalel       27          Tabarka     42500.0
6           Karim       22          Sousse      45000.0
7           Jamil       24          Hammamet    10000.0
```

## 8. SQLite - Fonctions utiles

SQLite possède de nombreuses fonctions intégrées pour effectuer le traitement de données sous forme de chaîne ou numériques.

Avant de commencer à donner des exemples des fonctions mentionnées ci-dessous, considérons le tableau EMPLOYEE avec les enregistrements suivants :

```
ID          NOM         AGE         ADRESSE     SALAIRE
----------  ----------  ----------  ----------  ----------
1           Nawel       32          Chebba              20000.0
2           Ali         25          Tabarka             15000.0
3           Tarak       23          Nabeul      20000.0
4           Mariam      25          Raoued      65000.0
5           Dalel       27          Tabarka             85000.0
6           Karim       22          Sousse              45000.0
7           Jamil       24          Hammamet            10000.0
```

Vous trouverez ci-dessous une liste de quelques fonctions intégrées SQLite utiles et toutes ne sont pas sensibles à la casse, ce qui signifie que vous pouvez utiliser ces fonctions en majuscule, en majuscule ou en mixte.

| Fonctions SQLite | Description de la fonction | Exemples |
|---|---|---|
| COUNT | La fonction d'agrégation SQLite COUNT est utilisée pour compter le nombre de lignes d'une table de base de données. | `SELECT count(*) FROM EMPLOYEE;`<br>Résultats :<br>`count(*)`<br>`----------`<br>`7` |
| MAX | La fonction d'agrégation SQLite MAX nous permet de sélectionner la valeur la plus élevée (maximum) pour une colonne donnée. | `SELECT max(SALAIRE) FROM EMPLOYEE;`<br>Résultats :<br>`max(SALAIRE)`<br>`-----------`<br>`85000.0` |
| MIN | La fonction d'agrégation SQLite MIN nous permet de sélectionner la valeur la plus basse (minimum) pour une colonne donnée. | `SELECT min(SALAIRE) FROM EMPLOYEE;`<br>Résultats :<br>`min(SALAIRE)`<br>`-----------` |

| | | |
|---|---|---|
| | | 10000.0 |
| AVG | La fonction d'agrégation SQLite AVG sélectionne la valeur moyenne pour certaines colonnes de la table. | `SELECT avg(SALAIRE) FROM EMPLOYEE;`<br>Résultats :<br>`avg(SALAIRE)`<br>`----------`<br>`37142.8571428572` |
| SUM | La fonction d'agrégation SQLite SUM permet de sélectionner le total d'une colonne numérique. | `SELECT sum(SALAIRE) FROM EMPLOYEE;`<br>Résultats :<br>`sum(SALAIRE)`<br>`----------`<br>`260000.0` |
| RANDOM | La fonction SQLite RANDOM renvoie un entier pseudo-aléatoire compris entre : 9223372036854775808 et +9223372036854775807. | `SELECT random() AS Random;`<br>Résultats :<br>`Random`<br>`----------`<br>`5876796417670984050` |
| ABS | La fonction ABS de SQLite renvoie la valeur absolue de l'argument numérique. | `SELECT abs(5), abs(-15), abs(NULL)`<br>Résultats :<br>`abs(5)  abs(-15)  abs(NULL)`<br>`----------  ----------  ----------`<br>`5  15` |
| UPPER | La fonction SQLite UPPER convertit une chaîne en lettres majuscules. | `SELECT upper(NOM) FROM EMPLOYEE;`<br>Résultats :<br>`upper(NOM)`<br>`----------`<br>`NAWEL`<br>`ALI`<br>`TARAK`<br>`MARIAM`<br>`DALEL`<br>`KARIM`<br>`JAMIL` |
| LOWER | La fonction SQLite LOWER convertit une chaîne en lettres minuscules. | `SELECT lower(NOM) FROM EMPLOYEE;`<br>Résultats :<br>`lower(NOM)`<br>`----------`<br>`Nawel`<br>`Ali` |

| | | |
|---|---|---|
| | | Tarak<br>Mariam<br>Dalel<br>Karim<br>Jamil |
| LENGTH | La fonction SQLite LENGTH renvoie la longueur d'une chaîne. | `SELECT NOM, length(NOM) FROM EMPLOYEE;`<br>Résultats :<br>NOM length(NOM)<br>---------- ------------<br>Nawel 4<br>Ali 5<br>Tarak 5<br>Mariam 4<br>Dalel 5<br>Karim 3<br>Jamil 5 |

Pour plus de détails, vous pouvez consulter la documentation officielle de SQLite : https://www.sqlite.org/