SQLITE 3

sqlite3-interface DB-API 2.0 pour la base de donnée SQLite

SQLite est une librairie C qui procure une base de données poids plume sur disque, qui ne nécessite pas l'existence d'un serveur de processus séparé et permet l'accès à une base de donnée en utilisant une variante non standard des langages de requête SQL. Certaines applications peuvent utiliser SQLite pour l'enregistrement interne de données. Il est aussi possible de prototyper une application en utilisant SQLite et d'en porter ensuite le code pour une base de données plus puissante telle que PostgreSQL ou Oracle.

`sqlite3` a été écrit pas Gerhard Häring et procure une interface **SQL** conforme avec les spécifications **DB-API 2.0** décrite par la **PEP 249** (*Python* Enhancement Proposals / proposition de nouvelles caractéristiques pour Python)

Pour utilise le **module** il faut d'abord créer un objet **`Connection`** qui représente la base de données. Dans ce qui suit, les données seront enregistrées dans le fichier **`/tmp/example`** :

```
conn = sqlite3.connect('/tmp/example')
```

On peut également utiliser le nom spécial **`:memory :`** pour créer une base de donnée en mémoire vive (RAM).

Une fois l'objet **`Connection`** obtenu, on crée un objet **`Cursor`** et on appelle la méthode **`execute()`** pour réalise les commandes SQL :

```
c = conn.cursor()

# création de table
c.execute('''create table stocks
             (date text, trans text, symbol text, qty real, price real)''')

# insérer une ligne de données
c.execute("""insert into stocks
             values ('2006-01-05','BUY','RHAT',100,35.14)""")

# sauvegarder (commit) les changements
conn.commit()

# on peut fermer l'ojbet cursor si on a fini de s'en servir
c.close()
```

Habituellement les opérations SQL auront besoin d'utiliser les valeurs de variables Python. Il faut éviter d'utiliser les opérations sur les chaînes Python parce que ce processus n'est pas sûr. Le programme est alors vulnérable aux attaques SQL.

À la place, on utilise les paramètres de substitution de l'API. Mettre **`?`** (point d'interrogation) comme place réservée pour utiliser une valeur, et ensuite produire un tuple des valeurs comme second argument de la méthode **`execute()`** de l'objet **`cursor`**. (D'autres modules de bases de données peuvent utiliser un symbole de place réservée différent, comme **`%s`** ou **`:1`**.). Par exemple :

```
# Ne jamais faire ceci -- code non sécurisé!
symbol = 'IBM'
c.execute(" select * from stocks where symbol = '%s'" % symbole)

# Faire ceci à la place
t = (symbole,)
c.execute('select * from stocks where symbol=?', t)

# Un exemple plus consistant
for t in [('2006-03-28', 'BUY', 'IBM', 1000, 45.00),
          ('2006-04-05', 'BUY', 'MSOFT', 1000, 72.00),
          ('2006-04-06', 'SELL', 'IBM', 500, 53.00),
         ]:
```

```
    c.execute('insert into stocks values (?,?,?,?,?)', t)
```

Pour retrouver les données après avoir exécuté une commande **SELECT**, on peut, soit traiter l'objet **`Cursor`** comme un itérateur (**`iterator`**), soit appeler la méthode **`fetchone()`** pour retrouver une ligne à la fois qui corresponde à la requête, soit appeler **`fetchall()`** pour récupérer une liste des lignes trouvées.

Voici un exemple utilisant la forme avec itérateur :

```
>>> c = conn.cursor()
>>> c.execute('select * from stocks order by price')
>>> for row in c:
...     print(row)
...
('2006-01-05', 'BUY', 'RHAT', 100, 35.14)
('2006-03-28', 'BUY', 'IBM', 1000, 45.0)
('2006-04-06', 'SELL', 'IBM', 500, 53.0)
('2006-04-05', 'BUY', 'MSOFT', 1000, 72.0)
>>>
```

voir aussi :
`http://code.google.com/p/pysqlite/`

La page web **pysqlite** est une page sqlite3 développé en externe sous le nom **"pysqlite"**

http://www.sqlite.org

C'est la page web de **SQLite** ; la documentation décrit la syntaxe et les types de données disponibles pour le dialecte SQL supporté.

PEP 249 - Database API Specification 2.0

La **PEP** a été écrite par Marc-André Lemburg.

1. Fonctions et constantes du module.

`sqlite3.PARSE_DECLTYPES`

Cette constante est destinée à être utilisée avec le paramètre **`detect_types`** de la fonction **`connect()`**.

Si on poser cette constante le module **sqlite3** analyse le type déclaré pour chaque colonne qu'il retourne. Le premier mot de type déclaré sera analysé, c'est à dire que pour "**integer primary key**" (clef primaire de type **`integer`**) l'analyse se fera sur "**integer**" ou encore pour "**number(10)**", "**number**" sera analysé. Puis pour cette colonne, le dictionnaire de conversion sera consulté et la fonction de conversion associée au type sera utilisé.

`sqlite3.PARSE_COLNAMES`

Cette constante est destinée à être utilisée avec le paramètre **`detect_types`** de la fonction **`connect().`**

Si on pose cette constante l'interface **`sqlite3`** analyse le nom de chaque colonne qu'elle retourne. Elle cherche une chaîne contenue en elle [*mytype*], et alors décide que "*mytype*" est le type de la colonne. Elle essaie de trouver une entrée "*mytype*" dans le dictionnaire de conversion et utilise la fonction de conversion associée pour retourner la valeur. Le nom de colonne trouvé dans le **`Cursor.description`** est seulement le premier mot du nom de colonne, c'est-à-dire que si on utilise quelque chose comme **`'as "x [datetime]"'`** dans le code SQL, alors l'analyse se fera sur tout jusqu'au premier blanc pour obtenir le nom de colonne : la colonne sera simplement nommée **`"x"`**.

`sqlite3.connect(`***`database[, timeout, detect_types, isolation_level, check_same_thread, factory, cached_statements]`***`)`

Ouvre une connexion à la base de donnée contenue dans le fichier ***`database`*** . On peut utiliser

":memory :" pour ouvrir une connexion sur une base de données résidant en mémoire vive au lieu d'un disque.

Quand on accède à une base de données par plusieurs connexions, et que l'un des processus modifie la base de données, la base SQLite est verrouillée jusqu'à ce que la transaction soit validée (commit). Le paramètre ***timeout*** spécifie le temps à attendre pour le verrouillage avant de déclencher une exception. Le temps par défaut pour le paramètre ***timeout*** est 5.0 (5 secondes)

Pour le paramètre **isolation_level** voir la propriété **Connection.isolation_level** des objets **Connection**.

D'un point de vue natif, SQLite connaît les types suivants : **TEXT, INTEGER, FLOAT, BLOB**, et **NULL.** Pour utiliser d'autres types, l'utilisateur doit ajouter lui-même leur support. Le paramètre ***detect_types*** et les **convertisseurs** utilisateurs sont enregistrée grâce à la fonction au niveau module **register_converter()** qui réalise cela facilement.

detect_types est par défaut à 0 (aucun type n'est détecté) ; On peut poser n'importe quelle combinaison de **PARSE_DECLTYPES** et **PARSE_COLNAMES** pour changer sa valeur et autoriser les détections de types.

Par défaut, le module **sqlite3** utilise sa classe **Connection** pour ses appels de connexion. On peut néanmoins sous-classer la classe **Connection** et disposer d'un **connect()** appartenant à la classe utilisateur en pourvoyant cette classe du paramètre **factory**.

Consulter la section SQLite et types Python de ce manuel pour les détails.

Le module sqlite3 utilise en interne un cache de commande qui évite à l'analyseur SQL de faire une analyse overhead (commandes définies avant le début du texte analysé) . Si on désire poser explicitement le nombre de commandes qui sont mises en caches lors de la connexion, il faut poser le paramètre **cached_statements**. La valeur par défaut est de 100 commandes.

sqlite3.register_converter(*typename, callable*)

Enregistre un ***callable*** pour convertir un *bytestring* (chaîne dont les éléments sont des bytes/octets) de la base de donnée en un type Python utilisateur. Le ***callable*** doit être invoqué pour toutes les valeurs de la base qui sont de type ***typename***. Il faut conférer le paramètre ***detect_type*** de la fonction **connect()** pour que la détection de type soit effective. Il faut évidemment que le ***typename*** utilisé et le nom du type dans la requête soient concordants.

sqlite3.register_adapter(*type, callable*)

Enregistre un ***callable*** pour convertir le type ***type*** utilisateur de Python en l'un des types connus de SQLite. Le ***callable*** prends comme unique paramètre la valeur Python et doit retourner une valeur de l'un des types suivants : **int, float, str** ou **byte**.

sqlite3.complete_statement(*sql*)

Retourne **True** si la chaîne ***sql*** contient une ou plusieurs commandes SQL terminées par un point-virgule. Il ne vérifie pas que le SQL est syntaxiquement correct, seulement qu'il n'y a aucun littéral chaîne enclos dans ***sql*** et que la commande se termine par un point-virgule.

Ceci peut peut être utilisé pour construire une commande shell pour SQLite comme dans l'exemple qui suit :

```
# Une commande shell minimale pour essayer
import sqlite3

con = sqlite3.connect(":memory:")
con.isolation_level = None
cur = con.cursor()
buffer = ""

print("Entrez votre commande SQL pour l'exécuter avec sqlite3.")
print("Entrez un ligne vide pour terminer.")
```

```
while True:
    line = input()
    if line == "":
        break
    buffer += line
    if sqlite3.complete_statement(buffer):
        try:
            buffer = buffer.strip()
            cur.execute(buffer)

            if buffer.lstrip().upper().startswith("SELECT"):
                print(cur.fetchall())
        except sqlite3.Error as e:
            print("An error occurred:", e.args[0])
        buffer = ""
```

con.close()

sqlite3.enable_callback_tracebacks(*flag*)

Par défaut, on ne peut pas produire de trace d'exécution (traceback) avec les fonction utilisateurs, aggrégats, convertisseurs, fonction callback d'autorisation etc. Si on veut les déboguer, on peut appeler cette fonction avec le ***flag*** à **True**. Ensuite, ont produit la trace d'exécution à partir des callbacks sur **sys.sterr**. Mettre le ***flag*** à **False** pour désactiver la commande.

2. Les objets Connection

class sqlite3.Connection

Une connexion à une base de donnée QSLite a les attributs et les méthodes qui suivent :

Connection.isolation_level

Fournit ou pose le niveau actuel d'isolement. C'est **None** pour le mode *autocommit* ou l'une des constantes "**DEFERRED**", "**IMMDIATE**" ou "**EXCLUSIVE**". Voir la section **Controlling Transaction** pour davantage d'explications.

Connection.in_transaction -v.3.2-

True si la transaction est active (il y a des changements non validés -*committed*-). **False** sinon. En lecture seulement.

Connection.cursor([*cursorClass*])

La méthode **cursor** accepte un seul paramètre optionnel ***cursorClass***. S'il est présent, il doit être un objet **cursor** utilisateur qui étend **sqlite3.Cursor**.

Connection.commit()

Cette méthode valide (*commit*) la transaction courante. Si un utilisateur n'appelle pas cette méthode, tout ce qui a été fait depuis le dernier appel de **commit()** est invisible depuis les autres connexions à la base de données. Si l'utilisateur s'étonne de ne pas voir les données qu'il a écrites sur la base, qu'il vérifie en priorité s'il n'a pas oublié d'appeler cette méthode.

Connection.rollback()

Cette méthode remonte toutes les modifications de la base depuis le dernier appel de **commit()**. Elle détruit le **commit()**.

Connection.close

Ferme la connexion à la base de données. Il faut noter que ceci n'est pas automatiquement fait en appelant **commit()**. Et si l'utilisateur ferme la base de données sans avoir d'abord appelé **commit()**, tous les changements seront perdus.

Connection.execute(*sql*[, *parameters*])

Ceci est court-circuit non standard qui crée un objet C**ursor** intermédiaire en appelant la méthode **cursor()**, puis appelle la méthode **execute()** avec les paramètres donnés.

Connection.executemany(*sql*[, *parameters*])

Ceci est court-circuit non standard qui crée un objet C**ursor** intermédiaire en appelant la méthode **cursor()**, puis appelle la méthode **executemany** avec les paramètres donnés.

Connection.executescript(*sql_script*)

Ceci est court-circuit non standard qui crée un objet C**ursor** intermédiaire en appelant la méthode **cursor()**, puis appelle la méthode **executscript** avec le paramètre donné.

Connection.create_function(*name, num_params, func*)

Crée une fonction définie par l'utilisateur qui peut ultérieurement être utilisée à l'intérieur de commandes SQL sous le nom ***name***. Le paramètre ***num_params*** est le nombre de paramètres que la fonction accepte et ***func*** est une fonction Python appelable, qui est appelée comme la fonction SQL.

La fonction peut retourner n'importe quel type connu de SQLite : **byte, str, int, float** et **None**.

Exemple :

```
import sqlite3
import hashlib

def md5sum(t):
    return hashlib.md5(t).hexdigest()

con = sqlite3.connect(":memory:")
con.create_function("md5", 1, md5sum)
cur = con.cursor()
cur.execute("select md5(?)", ("foo",))
print(cur.fetchone()[0])
```

Connection.create_aggregate(*name, num_params, aggregate_class*)

Crée une fonction **aggregate** défini par l'utilisateur.

La classe **aggregate** doit implémenter une méthode **stop**, qui accepte le nombre de paramètre fixé par ***num_params*** et une méthode ***finalize*** qui doit retourner le résultat final de **aggregate**.

La méthode **finalize** peut retourner tous les types connus de SQLite : **byte, str, int, float** et **None**.

Exemple:

```
import sqlite3

class MySum:
    def __init__(self):
        self.count = 0
```

```
    def step(self, value):
        self.count += value

    def finalize(self):
        return self.count

con = sqlite3.connect(":memory:")
con.create_aggregate("mysum", 1, MySum)
cur = con.cursor()
cur.execute("create table test(i)")
cur.execute("insert into test(i) values (1)")
cur.execute("insert into test(i) values (2)")
cur.execute("select mysum(i) from test")
print(cur.fetchone()[0])
```

Connection.create_collation(*name*, *callable*)

Crée un **collation** (*comparaison*) avec le nom ***name*** et ***callable***. Le ***callable*** doit avoir deux arguments chaînes. Il retourne -1 si le premier est plus petit que le second, 0 s'ils sont d'ordre égal et 1 si le second est plus grand. Noter que cette commande définit l'ordre de tri **(ORDER BY** en SQL) de façon que les comparaisons n'affectent pas les autres opérations SQL.

Noter également que le ***callable*** doit considérer ses paramètres comme **bytestring** Python, normalement encodé en UTF-8.

L'exemple suivant montre une comparaison définie par l'utilisateur qui trie à l'envers.

```
import sqlite3

def collate_reverse(string1, string2):
    if string1 == string2:
        return 0
    elif string1 < string2:
        return 1
    else:
        return -1

con = sqlite3.connect(":memory:")
con.create_collation("reverse", collate_reverse)

cur = con.cursor()
cur.execute("create table test(x)")
cur.executemany("insert into test(x) values (?)", [("a",), ("b",)])
cur.execute("select x from test order by x collate reverse")
for row in cur:
    print(row)
con.close()
```

Pour supprimer une comparaison, appeler **create_collation** avec **None** pour ***callable***.

```
con.create_collation("reverse", None)
```

Connection.interrupt()

On peut appeler cette méthode depuis un thread différent pour faire avorter n'importe quelle requête pouvant être exécutés sur la connexion. La requête sera interrompue et l'appelant (*caller*) produira une exception.

Connection.set_authorizer(*authorizer_callback*)

La routine enregistre un *callback*. Le *callback* est invoqué pour chaque tentative pour accéder à une colonne d'une table de la base de donnée. Le *callback* doit retourner un **SQLITE_OK** si l'accès est autorisé, **SQL_DENY** si l'ensemble de la commande SQL doit avorter, avec une erreur, et **SQLITE_IGNORE** si la colonne doit être traitée comme une valeur **NULL.** Ces constantes sont disponibles dans le module **sqlite3**.

Le premier argument du *callback* désigne la sorte d'opération qui est autorisée. Le second et le troisième argument doivent être des arguments ou **None**, en fonction du premier argument. Le quatrième argument est le nom de la base de données ("**main**", "**temp**", etc), si applicable. Le cinquième argument est le nom du trigger le plus profond ou la vue qui est responsable pour autoriser l'accès, ou **None** si l'autorisation d'accès provient directement du code SQL.

Voir la documentation de SQLite sur les valeurs possibles pour le premier argument et la signification des second et troisième arguments selon les valeurs du premier. Toutes les constantes nécessaire dont disponibles dans le module **sqlite3**.

Connection.set_progress_handler(*handler, n*)

Cette routine enregistre un *callback*. Le *callback* est invoqué toutes les ***n*** instructions exécutées par la machine virtuelle SQLite. C'est utile si on désire être renseigné par **SQLite** pendant au cours des opérations qui durent longtemps, par exemple la mise à jour d'un GUI (Graphic Unit Interface).

Si on désire supprimer un tel handler de progression préalablement installé, appeler la routine avec **None** comme ***handler***.

Connection.enable_load_extension(*enabled*) -version 3.2.-

Cette routine permet ou interdit à la machine **SQLite** de charger des extensions **SQLite** à partir de librairies partagées. Les extensions **SQLite** peuvent définir de nouvelles fonctions, agrégats, ou l'implémentation complète de nouvelles tables virtuelles. Un extension bien connue est l'extension "*recherche dans tout le texte*" qui est distribuée avec **SQLite**.

```
import sqlite3

con = sqlite3.connect(":memory:")

# rendre le chargement d'extension disponible
con.enable_load_extension(True)

# Charger l'extension "fulltext search"
con.execute("select load_extension('./fts3.so')")

#
on peut également charger une extension par le biais de l'API
# con.load_extension("./fts3.so")

# interdire toute nouvelle extension
con.enable_load_extension(False)

# exemple de SQLite wiki
con.execute("create virtual table recipe using fts3(name, ingredients)")
con.executescript("""
    insert into recipe (name, ingredients) values ('broccoli stew',
                        'broccoli peppers cheese tomatoes');
    insert into recipe (name, ingredients) values ('pumpkin stew',
                        'pumpkin onions garlic celery');
    insert into recipe (name, ingredients) values ('broccoli pie',
```

```
                        'broccoli cheese onions flour');
    insert into recipe (name, ingredients) values ('pumpkin pie',
                        'pumpkin sugar flour butter');
    """)
for row in con.execute("select rowid, name,
                  ingredients from recipe where name match 'pie'"):
    print(row)
```

Les extensions chargeables sont non disponibles par défaut (Note ci-dessous).

Connection.load_extension(*path*) -version 3.2.-

Cette routine charge une extension **SQLite** à partir d'une librairie partagée. Pour rendre disponible cette extension exécuter **enable_load_extension()** avant d'utiliser cette routine.

Les extensions chargeables sont non disponibles par défaut.

Note sur les extensions chargeables : Le module sqlite3 n'est pas construit avec les extensions chargeables par défaut, parce que certaines plates formes (Mac OS X) ont une librairie SQLite compilée sans cette disposition. Pour disposer du support des extensions chargeables, il faut passer –**enable-loadable-sqlite-extensions** dans la configuration.

Connection.row_factory

On peut changer cet attribut en lui affectant un *callable* qui accepte en paramètre **cursor** et ligne originale sous forme d'un tuple et retourne la ligne résultante réelle. Ainsi, on peut implémenter une façon plus satisfaisante de retourner des résultats, comme retourner un objet qui peut aussi accéder aux colonnes par leur nom.

Exemple :

```
import sqlite3

def dict_factory(cursor, row):
    d = {}
    for idx, col in enumerate(cursor.description):
        d[col[0]] = row[idx]
    return d

con = sqlite3.connect(":memory:")
con.row_factory = dict_factory
cur = con.cursor()
cur.execute("select 1 as a")
print(cur.fetchone()["a"])
```

Si retourner un tuple ne suffit pas et qu'on désire un accès au colonnes par noms, on peut envisager de poser **row_factory** au type hautement optimisé **sqlite3.Row**.

Row procure à la fois un accès par index et par nom -insensible à la casse- avec en plus absence de mémorisation préalable (*overhead*). C'est probablement meilleur qu'une approche par dictionnaire utilisateur ou même qu'une solution fondée sur les lignes de bases de données (*db_row based solution*).

Connection.text_factory

On utilise cet attribut pour contrôler quels objets sont retournés par le type de données **TEXT**. Par défaut, cet attribut est posé à **str** et le module **sqlite3** retourne des objets **Unicode** pour **TEXT**. Si on désire retourner des *bytestrings* à la place, on peut poser l'attribut à **bytes.**

Pour des raisons d'efficacité, il y a également une façon de retourner des objets **str** uniquement pour les données non ASCII, et des objets **bytes** sinon. Pour activer cette disposition, poser l'attribut à

sqlite3.OptimizedUnicode.

On peut aussi poser l'attribut à n'importe quel *callable* qui accepte un unique paramètre **bytestring**, et retourne l'objet résultant. Voir l'exemple qui suit :

```
import sqlite3

con = sqlite3.connect(":memory:")
cur = con.cursor()

# Créer la table
con.execute("create table person(lastname, firstname)")

AUSTRIA = "\xd6sterreich"

# par défautn les lignes sont retournées en Unicode
cur.execute("select ?", (AUSTRIA,))
row = cur.fetchone()
assert row[0] == AUSTRIA

# mais sqlite3 peut toujours returner des bytestrings ...
con.text_factory = str
cur.execute("select ?", (AUSTRIA,))
row = cur.fetchone()
assert type(row[0]) == str
# les bytestrings peuvent encodés en UTF-8 sauf pour les
# enregistrement dans la base

assert row[0] == AUSTRIA.encode("utf-8")

# on peut aussi impémenter une fabrique de texte utilisateur
# en voici une qui ignore l'Unicode qui ne peut pas être
# décodé à partir de l'UTF-8

con.text_factory = lambda x: str(x, "utf-8", "ignore")
cur.execute("select ?",
            ( "Ceci est du latin-1 qui devrait mettre en erreur" +
             "\xe4\xf6\xfc".encode("latin1"),))
row = cur.fetchone()
assert type(row[0]) == str

# sqlite3 comporte une fabrique de texte optimisée qui retourne
# des objets bytestrings si les données sont en ASCII et
# sinon retourne de l'Unicode

con.text_factory = sqlite3.OptimizedUnicode
cur.execute("select ?", (AUSTRIA,))
row = cur.fetchone()
assert type(row[0]) == str

cur.execute("select ?", ("Germany",))
row = cur.fetchone()
assert type(row[0]) == str
```

Connection.total_changes

Retourne le nombre total de lignes de la base de données qui ont été modifiées, insérées ou effacées depuis le début de la connexion à la base de sonnées.

Connection.iterdump

Retourne un iterateur qui décharge la base de donnée en un texte au format SQL. Utile quand on sauvegarde une base de donnée en mémoire pour une restauration ultérieure. Cette fonction donne les mêmes possibilités sue la commande **.dump** du shell **sqlite3**.

Exemple :

```
#  Conversion du fichier mabase.db en fichier dump.sql

import sqlite3, os

con = sqlite3.connect('mabase.db')
with open('dump.sql', 'w') as f:
    for line in con.iterdump():
        f.write('%s\n' % line)
```

3. Les objets Cursor

class sqlite3.Cursor

Une instance de **Cursor** possède les attributs et méthodes qui suivent.

Cursor.execute(*sql*[, *parameters*])

Ecécute une commande SQL. La commande SQL peut être paramétrée, c'est-à-dire contenir des "*réserves de place*" (placeholders) au lieu de littéraux. Le module sqlite3 supporte deux sortes de "*réserves de place*" : les marqueurs ? (point d'interrogation / *style qmark*) et les "*réserves de place*" nommées (style nommé)

L'exemple qui suit montre comment utiliser le style qmark :

```
import sqlite3

con = sqlite3.connect("mydb")

cur = con.cursor()

who = "Yeltsin"
age = 72

cur.execute("select name_last" +
            ", age from people where name_last=? and age=?",
            (who, age))
print(cur.fetchone())
```

Cet exemple montre comment utiliser le style nommé

```
import sqlite3

con = sqlite3.connect("mydb")

cur = con.cursor()

who = "Yeltsin"
age = 72

cur.execute("select name_last'+
```

```
            ", age from people where name_last=:who and age=:age",
            {"who": who, "age": age})
print(cur.fetchone())
```

execute() n'exécute qu'une commande simple. Si on essaie d'exécuter plus d'une commande avec lui, il émet un avertissement (Warning). Utiliser **executescript()** si on désire exécuter plusieurs commandes SQL avec un seul appel.

Cursor.executemany(*sql, seq_of_parameters*)

Exécute une commande SQL avec toutes les séquences de paramètres ou les mappings trouvés dans la séquence ***sql.*** **L**e module sqlite3 permet une allocation utilisant un itérateur soumettant les paramètres, à la place d'une séquence.

Exemple avec itérateur :

```
import sqlite3

class IterChars:
    def __init__(self):
        self.count = ord('a')

    def __iter__(self):
        return self

    def __next__(self):
        if self.count > ord('z'):
            raise StopIteration
        self.count += 1
        return (chr(self.count - 1),) # this is a 1-tuple

con = sqlite3.connect(":memory:")
cur = con.cursor()
cur.execute("create table characters(c)")

theIter = IterChars()
cur.executemany("insert into characters(c) values (?)", theIter)

cur.execute("select c from characters")
print(cur.fetchall())
```

Court exemple avec un générateur :

```
import sqlite3

def char_generator():
    import string
    for c in string.letters[:26]:
        yield (c,)

con = sqlite3.connect(":memory:")
cur = con.cursor()
cur.execute("create table characters(c)")

cur.executemany("insert into characters(c) values (?)", char_generator())

cur.execute("select c from characters")
print(cur.fetchall())
```

Cursor.executescript(*sql_script*)

Ceci est une méthode non standard de convenance pour exécuter des commandes **SQL** multiples en une seule fois. Elle commence par émettre un commande **COMMIT**, puis elle exécute le script **SQL** donné en paramètre. ***sql_script*** doit être une instance de **str** ou **bytes**.

Exemple :

```
import sqlite3

con = sqlite3.connect(":memory:")
cur = con.cursor()
cur.executescript("""
    create table person(
        firstname,
        lastname,
        age
    );

    create table book(
        title,
        author,
        published
    );

    insert into book(title, author, published)
    values (
        'Dirk Gently''s Holistic Detective Agency',
        'Douglas Adams',
        1987
    );
    """)
```

Cursor.fetchone()

Cherche la ligne suivante d'un ensemble de résultats d'une requête, et retourne une seule séquence ou **None** quand aucune donnée n'est disponible.

Cursor.fetchmany([*size=cursor.arraysize*])

Recherche l'ensemble des lignes qui suivent du résultat d'une requête, et retourne une liste. La liste est vide si aucune ligne n'est disponible.

Le nombre de lignes à rechercher à chaque appel est spécifié par le paramètre ***size***. S'il n'est pas spécifié, la taille du tableau de l'objet C**ursor** définit le nombre de lignes à chercher. La méthode n'est pas obligée de rechercher le nombre de lignes indiqué par le paramètre : s'il reste moins de lignes que spécifié dans le paramètre, il se peut que la liste retournées comporte moins de ligne que le nombre souhaité.

Noter que le paramètre ***size*** doit être choisi à partir de considérations de performance. Pour une performance optimale, il est souvent préférable d'utiliser l'attribut ***arraysize***. Si le paramètre ***size*** est explicité, alors il est préférable d'utiliser la même valeur pour les appels de **fetchmany()** successifs.

Cursor.fetchall()

Recherche toutes les lignes disponibles du résultat d'une requête et en retourne la liste. Noter que l'attribut ***cursor.arraysize*** peut affecter les performances de l'opération. Une ligne vide est retournée quand aucune ligne n'est disponible.

`Cursor.rowcount`

Bien que la classe **`Cursor`** du module **`sqlite3`** implémente cet attribut, la machine base de donnée a son propre support pour la détermination de "lignes affectées"/"lignes sélectionnées" qui est plus original.

Pour la commande **DELETE**, SQLite met **`rowcount`** à 0 si on fait **DELETE FROM** ***table*** sans condition.

Pour la commande **`executemany()`**, le nombre des modifications est cumulé dans **`rowcount`**.

Comme il est requis par les spécification **Python DB API**, l'attribut **`rowcount`** "*est -1 dans le cas ou aucun* ***`executeXX()`*** *n'a été appelé sur l'objet* ***`cursor`***, *ou si le* ***`rowcount`*** *de la dernière opération n'est pas déterminable par l'interface*".

Ceci inclut les commandes **SELECT** parce que l'on ne peut savoir le nombre de lignes qu'une telle requête a produit avant que toutes les lignes aient été recherchées.

`Cursor.lastrowid`

Cet attribut en lecture seule donne l'id (***rowid***) de la dernière ligne modifiée. Cette valeur est posée uniquement par la réalisation d'une commande **INSERT** passant par la méthode **`execute()`**. Pour les opérations autres que **INSERT**, ou quand **`executemany()`** est appelée, **`lastrowid`** est posé à **`None`**.

`Cursor.description`

Cet attribut en lecture seule donne les noms des colonnes de la dernière requête. Par compatibilité avec **Python DB API**, il retourne un tuple de 7 éléments pour chaque colonne, mais avec au delà du premier élément, 6 items à **`None`**.

Cet attribut est posé pour les commandes **SELECT** sans aucun souci de concordance avec des lignes.

4. Les objets Row

`class sqlite.Row`

Une instance de **`Row`** sert à optimiser fortement **`row_factory`** pour les objets **`Connection`**. Il essaie d'imiter un tuple dans la plupart de ses caractéristiques.

Elle supporte l'accès *mapping* par nom de colonne et index, itération, représentation, test d'égalité et longueur (**`len()`**)

Si deux objets **`Row`** on exactement les mêmes colonnes et que leurs membres sont égaux, ils sont considérés comme égaux dans une comparaison.

`keys()`

Cette méthode retourne un tuple des noms de colonnes. Immédiatement après une requête, elle est le premier membre de chaque tuple dans **`Cursor.description`**.

On suppose qu'une table est initialisée comme dans l'exemple qui précède :

```
conn = sqlite3.connect(":memory:")
c = conn.cursor()
c.execute('''create table stocks
    (date text, trans text, symbol text, qty real, price real)''')
c.execute("""insert into stocks
          values ('2006-01-05','BUY','RHAT',100,35.14)""")
conn.commit()
c.close()
```

Maintenant on insère `Row` :

```
>>> conn.row_factory = sqlite3.Row
>>> c = conn.cursor()
>>> c.execute('select * from stocks')
<sqlite3.Cursor object at 0x7f4e7dd8fa80>
>>> r = c.fetchone()
>>> type(r)
<class 'sqlite3.Row'>
>>> tuple(r)
('2006-01-05', 'BUY', 'RHAT', 100.0, 35.14)
>>> len(r)
5
>>> r[2]
'RHAT'
>>> r.keys()
['date', 'trans', 'symbol', 'qty', 'price']
>>> r['qty']
100.0
>>> for member in r:
...     print(member)
...
2006-01-05
BUY
RHAT
100.0
35.14
```

5. SQLite et types Python

5.1. Introduction

SQLite supporte en natif les types suivants : **NULL, INTEGER, REAL, TEXT, BLOB.**

Les types Python qui suivent peuvent être utilisés dans SQLite sans problème :

type Python	type SQLite
None	NULL
int	INTEGER
float	REAL
str	TEXT
bytes	BLOB

SQLite convertit par défaut les types Python ci-dessous :

type SQLite	type Python
NULL	None
INTEGER	int
REAL	float

type SQLite	type Python
TEXT	cela dépend dans **text_factory**, avec **str** by default
BLOB	**bytes**

Le système des types du module **sqlite3** est extensible de deux façons :

- on peut enregistrer des type Python additionnels dans une base **SQLite**, via des objets adaptateurs,

- et on peut le module sqlite3 qui convertit les types **SQLite** en divers type Python via les convertisseurs (**converter**).

5.2. adaptateurs pour enregistrer des types Python dans une base SQLite

Comme vu auparavant, SQLite ne supporte qu'un nombre limité de types natifs. L'emploi d'autres type Python avec SQLite nécessite leur adaptation à l'un des types supporté par SQLite, c'est-à-dire un des types suivante : **None, int, float, str, byte**.

Le module **sqlite3** utilise des objets d'adaptation à Python, suivant en cela la **PEP 246**. Le protocole à utiliser est **PrepareProtocol**.

Il y a deux façons de rendre capable la module sqlite3 d'adapter un type utilisateur Python à l'un de ceux supportés :

- laisser l'objet s'adapter de lui-même. C'est une bonne approche lorsque c'est l'utilisateur final qui écrit les classes. Voici un exemple :

```
class Point:
    def __init__(self, x, y):
        self.x, self.y = x, y
```

Maintenant, on veut enregistrer le point par une unique colonne **SQLite**. En premier, choisir l'un des types supporté pour représenter le point. On peut simplement utiliser un type **str**, avec le point-virgule comme séparateur des coordonnées. On a alors besoin d'ajouter à la classe une méthode **__conform__(self, protocol)** qui doit retourner la valeur convertie. Le protocole de paramétrage est **PrepareProtocol**.

```
import sqlite3

class Point:
    def __init__(self, x, y):
        self.x, self.y = x, y
    def __conform__(self, protocol):
        if protocol is sqlite3.PrepareProtocol:
            return "%f;%f" % (self.x, self.y)

con = sqlite3.connect(":memory:")
cur = con.cursor()

p = Point(4.0, -3.2)
cur.execute("select ?", (p,))
print(cur.fetchone()[0])
```

- **enregistrer un adaptateur callable**. L'autre possibilité consiste à créer une fonction qui convertit la type et représentation "chaîne" et enregistrer la fonction avec.

register_adapter().

```
import sqlite3

class Point:
    def __init__(self, x, y):
        self.x, self.y = x, y

def adapt_point(point):
    return "%f;%f" % (point.x, point.y)

sqlite3.register_adapter(Point, adapt_point)

con = sqlite3.connect(":memory:")
cur = con.cursor()

p = Point(4.0, -3.2)
cur.execute("select ?", (p,))
print(cur.fetchone()[0])
```

Le module **sqlite3** possède deux adaptateurs par défaut pour les types internes (buit-in) **datetime.date** et **datetime.datetime**. Maintenant, si on désire enregistrer des objets **datetime.datetime** non en représentation **ISO** mais en mode *timestamp* d'**Unix :**

```
import sqlite3
import datetime
import time

def adapt_datetime(ts):
    return time.mktime(ts.timetuple())

sqlite3.register_adapter(datetime.datetime, adapt_datetime)

con = sqlite3.connect(":memory:")
cur = con.cursor()

now = datetime.datetime.now()
cur.execute("select ?", (now,))
print(cur.fetchone()[0])
```

5.3. Conversion de valeurs SQLite en types utilisateur Python

Vouloir écrire un adaptateur convertissant des types SQLite en type Python, suppose que des types Python ont été enregistrés en types **SQLite**. Mais pour qu'un adaptateur soit réellement efficace, il faut que la boucle de conversion Python ==> SQLite ==> Python soit un travail nul !

Entrer les convertisseurs.

On revient sur la classe Point. On a enregistré les coordonnées x et y dans une chaîne SQLite, en les séparant par un point-virgule. En premier on définit une fonction convertisseur qui accepte la chaîne comme paramètre et construit un objet **Point** correspondant.

Note. La fonction de conversion doit toujours être appelée avec une chaîne, indépendamment de la façon dont elle a été envoyée à SQLite.

```
def convert_point(s):
    x, y = map(float, s.split(";"))
    return Point(x, y)
```

Maintenant on a besoin de faire que le module sqlite3 sache ce que ce que l'on a sélectionné dans la base est actuellement un **`Point`**. Il y a deux façons de faire :

* implicitement par un type déclaré
* explicitement par le nom de la colonne

Les deux façons de faire sonr décrits da la section Module functions ans constants, dans les entrées concernant les constantes **PARSE_DECLTYPES** et **PARSE_COLNAMES**

L'exemple suivant illustre les deux approches :

```
import sqlite3

class Point:
    def __init__(self, x, y):
        self.x, self.y = x, y

    def __repr__(self):
        return "(%f;%f)" % (self.x, self.y)

def adapt_point(point):
    return "%f;%f" % (point.x, point.y)


def convert_point(s):
    x, y = list(map(float, s.split(";")))
    return Point(x, y)

# Enregistrer l'adaptateur
sqlite3.register_adapter(Point, adapt_point)

# Enregistrer le convertisseur
sqlite3.register_converter("point", convert_point)

p = Point(4.0, -3.2)

#########################
# 1) Utiliser les types déclarés
con = sqlite3.connect(":memory:", detect_types=sqlite3.PARSE_DECLTYPES)
cur = con.cursor()
cur.execute("create table test(p point)")

cur.execute("insert into test(p) values (?)", (p,))
cur.execute("select p from test")
print("with declared types:", cur.fetchone()[0])
cur.close()
con.close()

#######################
# 1) Utilise les noms de colonnes
con = sqlite3.connect(":memory:", detect_types=sqlite3.PARSE_COLNAMES)
cur = con.cursor()
cur.execute("create table test(p)")

cur.execute("insert into test(p) values (?)", (p,))
cur.execute('select p as "p [point]" from test')
print("with column names:", cur.fetchone()[0])
cur.close()
con.close()
```

5.4. Adaptateurs et convertisseurs par défaut.

Il exite des adaptateur par défaut pour les types **`date`** et **`datetime`** du module **`datetime`**. Ils sont envoyés pour envoyer **dates** et **timestamps ISO** à **SQLite**. Les convertisseurs par défaut sont enregistrés sous le nom "**date**" pour **`datetime.date`** et sous le nom de "**timestamps**" pour **`datetime.datetime`**.

De cette façon, on peut utiliser **date/datetime** depuis Python sans bricolage additionnel dans la plupart des cas. Le format des adaptateurs est aussi compatible avec les fonctions expérimentales **date/time** de **SQLite**.

L'exemple suivant en fait la démonstration :

```
import sqlite3
import datetime

con = sqlite3.connect(":memory:", detect_types=sqlite3.PARSE_DECLTYPES|
sqlite3.PARSE_COLNAMES)
cur = con.cursor()
cur.execute("create table test(d date, ts timestamp)")

today = datetime.date.today()
now = datetime.datetime.now()

cur.execute("insert into test(d, ts) values (?, ?)", (today, now))
cur.execute("select d, ts from test")
row = cur.fetchone()
print(today, "=>", row[0], type(row[0]))
print(now, "=>", row[1], type(row[1]))

cur.execute('select current_date as "d [date]", current_timestamp as "ts
[timestamp]"')
row = cur.fetchone()
print("current_date", row[0], type(row[0]))
print("current_timestamp", row[1], type(row[1]))
```

6. Contrôle des transactions

Par défaut, le module sqlite3 ouvre implicitement des transactions avant une commande de DML (Data Modification Langage : **INSERT/UPDATE/DELETE/REPLACE**), et conclut (commit) les transactions implicitement avant une commande non DML ou non requête (autre que **SELECT**/ **INSERT/UPDATE/DELETE/REPLACE).**

Aussi si on est à l'intérieur d'une transaction et que vous exécutez une commande comme **CREATE TABLE ..., VACUUM, PRAGMA**, le module **`sqlite3`** va conclure (commit) implicitement avant d'exécuter cette commande. Il y a deux raisons pour cela. La première est que certaines de ces commande ne peuvent se réaliser à l'intérieur d'une transaction. L'autre raison est que sqlite3 a besoin de garder la trace de l'état de la transaction (transaction active ou pas). L'état courant d'une transaction est contenu dans l'attribut **`Connection.in_transaction`** de l'objet **`Connection`**.

On peut contrôler quelle sorte de commande **BEGIN** sqlite3 exécute implicitement (ou pas du tout) à travers le paramètre **`isolation_level`** à l'appel de **`connect()`**, ou par le propriété **`isolation_level`** des connexions.

Si on préfète le mode **autocommit**, alors l'attribut **`isolation_level`** est posé à **`None`**.

Sinon, laisser la valeur par défaut, qui produit une vraie commande "**BEGIN**", ou utiliser l'un des niveaux d'isolement supporté par **SQLite** : "**DEFERRED**", "**IMMEDIATE**" or "**EXCLUSIVE**"

7. Être efficace avec sqlite3

7.1. Utilise les méthodes court-circuit

En utilisant les méthodes non standard **`execute(), executemany() et executescript()`** de l'objet **Connection**, le code sera plus concis car on n'a pas à créer explicitement les objets **`Cursor`** -souvent

superflus-. À la place les objets **Cursor** sont créés implicitement et ces méthode court-circuit retournent des objets **Cursor**. Ainsi, pour exécuter une commande **SELECT** et itérer dessus directement on n'utilise qu'un simple appel sur l'objet **Connection**.

```
import sqlite3

persons = [
    ("Hugo", "Boss"),
    ("Calvin", "Klein")
    ]

con = sqlite3.connect(":memory:")

# Créer la table
con.execute("create table person(firstname, lastname)")

# Remplir the table
con.executemany("insert into person(firstname, lastname) values (?, ?)",
persons)

# Afficher les contnus de la table
for row in con.execute("select firstname, lastname from person"):
    print(row)

#  Utiliser une clause WHERE factice pour que SQLite n'utilise
#  pas le court-circuit delete qui efface les tables
print("I just deleted",
        con.execute("delete from person where 1=1").rowcount,
       "rows")
```

7.2. Accéder aux colonnes par nom et non par index

D'abord, utiliser la disposition du module sqlite3, sqlite3.Row, conçue pour être utilisée comme fabrique de lignes.

Les lignes "enveloppée" (wrapped) avec cette classe peuvent être accessibles à la fois par index (comme tuples) et par noms (insensibles à la casse).

```
import sqlite3

con = sqlite3.connect("mydb")
con.row_factory = sqlite3.Row

cur = con.cursor()
cur.execute("select name_last, age from people")
for row in cur:
    assert row[0] == row["name_last"]
    assert row["name_last"] == row["nAmE_lAsT"]
    assert row[1] == row["age"]
    assert row[1] == row["AgE"]
```

7.3 Utiliser la connexion comme un gestionnaire de contexte

Les objets **Connection** peuvent être utilisés comme gestionnaire de contexte qui réalise automatiquement les transactions de *commit* et de *rollback*. S'il se produit une exception, la transaction est remontée (*rollback*) sinon elle est validée (*commit*).

```
import sqlite3
```

```
con = sqlite3.connect(":memory:")
con.execute("create table person (id integer primary key,"
            + "firstname varchar unique)")

# Succès : con.commit() est appelé automatiquement ensuite
with con:
    con.execute("insert into person(firstname) values (?)", ("Joe",))

# con.rollback() est appelé après la fin du bloc with avec
# une exception qui est levée et doit être attrapée
try:
    with con:
        con.execute("insert into person(firstname) values (?)",
                    ("Joe",))
except sqlite3.IntegrityError:
    print("on ne peut ajouter deux fois Joe")
```

8 Disposition communes

8.1. Multithreading

Les anciennes versions de SQLite avaient la possibilité de partager les connexion entre threads. C'est pourquoi le module Python désalloue le partage des objets **`Connection`** et **`Cursor`** entre threads. Si on essaie de partager, on provoque une exception à l'exécution.

Le seul cas dérogatoire et l'appel de la méthode **`interrupt()`**, qui n'a de sens que si elle peut effectivement être appelée depuis des threads différents.

Note sur les extensions chargeables : Le module sqlite3 n'est pas construit avec les extensions chargeables par défaut, parce que certaines plates formes (Mac OS X) ont une librairie SQLite compilée sans cette disposition. Pour disposer du support des extensions chargeables, il faut passer –**`enable-loadable-sqlite-extensions`** dans la configuration.

****
Si l'on se réfère à la documentation officielle, sont *callables* :

- Les fonctions et méthodes que nous définissons (*user-defined*)
- Les fonctions et méthodes natives (*built-in*)
- Les fonctions *generator*
- Les classes *new-style* ou *old-style* (nommées aussi *classic*)
- Les instances de classe possédant une méthode **`__call__`**

****

Une transaction permet de grouper plusieurs actions SQL et de les rendre atomiques (indissociables). Cet aspect est très important car bien souvent la mise à jour de données ne peut se faire avec un seul ordre SQL. Le **Commit** permet de valider l'ensemble des ordres SQL passés dans la transaction (les ordres de mise à jour ou d'insertion de données). Ce n'est qu'une fois cet ordre exécuté que les données sont considérées comme étant cohérentes (et donc écrites) dans la base.
Le **RollBack** permet lui au contraire d'annuler tous les ordres SQL passés dans la transaction.

Types de données dans la version SQLite Version 3

La plupart des machines SQL (toutes les machine SQL autres que **SQLite**, aussi loin que nous remontions) utilisent un typage statique rigide. Dans le typage statique, le type de donnée d'une valeur est déterminée par son conteneur -la colonne particulière dans laquelle la valeur est enregistrée.

SQLite utilise un système de typage dynamique plus général. Dans **SQLite**, le type de donnée d'une valeur est associée à la valeur elle-même, pas à son conteneur. Le système de typage dynamique de **SQLite** est rétro-compatible avec les systèmes statiques le plus connus dans les autres machines en ce sens que les formulations mise en œuvre sur les bases de données typées statiquement sont mises en œuvre de façon identiques dans **SQLite**. Néanmoins, le typage dynamique de **SQLite** permet de faire des choses qui ne sont pas possibles dans les bases de données traditionnelles typées de façon rigide.

1. Classes d'Enregistrement et Types de Données.

Chaque valeur enregistrée dans une base **SQLite** (ou manipulée par la machinerie de la base) a l'une des **Classes d'enregistrement** qui suit :

* **NULL** La valeur est une valeur NULL.

* **INTEGER**. La valeur est un entier signé ; enregistré dans 1, 2, 3, 4, 6 ou 8 octets selon la grandeur de la valeur.

* **REAL** La valeur est une valeur point flottant, enregistré sur 8 octets selon la norme IEEE : 64 bits : 1 bit de signe, 11 bits d'exposant (-1022 à 1023), 52 bits de mantisse.

* **TEXT**. La valeur est une chaîne, enregistrée selon l'encodage retenu pour la base (UTF-8, UTF-16BE ou UTF-16LE).

* **BLOB**. La valeur est une donnée blob **(*Binary Large OBject*),** enregistrée exactement comme elle a été entrée (permet d'enregistrer une image par exemple).

Noter que la **classe d'enregistrement** est un peu plus générale qu'un **type de données**. La classe d'enregistrement **INTEGER**, par exemple, inclut 6 formats d'entiers, de longueurs différentes. Ceci est répercuté dans les données du disque. Mais dès qu'une valeur **INTEGER** est lue sur le disque ou dans la mémoire pour un calcul, elle est convertie dans le type le plus général, en 64 bits signé. Et ainsi pour la plus grande part, la "classe d'enregistrement" ne peut se distinguer du type de données, et les deux termes sont habituellement interchangeables.

Toute colonnes d'une base **SQLite**, à l'exception d'une colonne **INTEGER PRIMARY KEY**, peut être utilisée pour enregistrer une valeur de n'importe quelle classe d'enregistrement.

Toute valeur spécifiée en SQL, qu'elle soit un littéral incluse dans une expression SQL, qu'elle soit un paramètre lié à une expression précompilée SQL a une classe d'enregistrement implicite. Selon les circonstances décrites ci-dessous, la machinerie de la base de donnée peut convertir les valeurs entre classes d'enregistrement (**INTEGER** et **REAL**) et **TEXT** au cours de l'exécution d'une requête.

1.1. Type de donnée booléen

SQLite n'a pas de classe d'enregistrement spécifique pour les booléens. Les valeurs booléennes sont enregistrées comme entiers, 0 (*false*) et 1 (*true*).

1.2. Types de donnée Date et Time

SQLite n'a pas de classe d'enregistrement destinées à enregistrer dates et durées. À la place, les fonction intégrée **`Date`** et **`Time`** de **SQLite** peuvent enregistrer dates et durées comme valeurs **TEXT**, **REAL** ou **INTEGER**.

* **TEXT** selon la norme ISO8601 (**"YYYY-MM-DD HH:MM:SS.SSS"**

* **REAL** comme nombre de jours julien depuis midi du 24 novembre de 4714 av. J.C. selon le "*proleptic Gregorian calendar*", (calendrier grégorien corrigé), notre calendrier courant.

* **INTEGER** comme temps Unix nombre de secondes écoulée depuis le premier janvier 1970 à 0 heures.

Les applications peuvent choisir d'enregistrer dates et durées dans n'importe lequel de ces formats, et le convertir librement entre les formats en utilisant les fonction incluses de date et durée.

2. Les affinités de type

Dans le but de maximiser la compatibilité entre **SQLite** et les autres gestionnaires de bases de données, **SQLite** comporte le concept "d'affinité de type" portant sur les colonnes. L'affinité de type d'une colonne est le type recommandé pour l'enregistrement des données de la colonne. L'idée importante ici, c'est que le type est **recommandé**, et non **requis**. Toute colonne peut toujours enregistrer n'importe quel type de données. Plus précisément, qu'une colonne, si un choix est donné, va préférer utiliser une classe d'enregistrement plutôt qu'une autre. La classe d'enregistrement préférée pour une colonne est appelée son "affinité".

À chaque colonne sans une base **SQLite** 3 est assignée l'un des affinités de type suivantes :

TEXT, NUMERIC, INTEGER, REAL, NONE

Une colonne d'affinité **TEXT** enregistre toutes les donnée en utilisant les classes d'enregistrement NULL, TEXT, BLOB. Si une donnée numérique est insérée sans une colonne d'affinité TEXT, elle est convertie en format texte avant l'enregistrement.

Une colonne avec l'affinité **NUMERIC** peut contenir des valeurs utilisant les cinq types d'enregistrement. Quand une donnée texte est insérée dans une colonne **NUMERIC**, la classe d'enregistrement du texte est convertie en **INTEGER** ou **REAL** (ordre préférentiel) si une telle conversion est sans perte d'information et réversible. Pour les conversions entre les classes d'enregistrement **TEXT** et **REAL**, **SQLite** considère que la conversion est sans perte et réversible si les 15 chiffres décimaux signifiants du nombre sont conservés. Si la conversion sans perte de **TEXT** vers **INTEGER** ou **REAL** n'est pas possible, alors le valeur est enregistrée dans une classe d'enregistrement **TEXT**. Aucune tentative n'est faite pour convertir en valeur **BLOB** ou **NULL**.

Une chaîne peut ressembler à un littéral point flottant, avec point décimal et/ou notation exponentielle, mais sa valeur peut être exprimée sous forme d'un entier : l'affinité **NUMERIC** la convertit en un entier. Ainsi, la chaîne **`'3.0e+5'`** est enregistrée dans une colonne ayant l'affinité **NUMERIC** comme l'entier **`300000`**, et pas la valeur en point flottant **`300000.0`**.

Une colonne qui utilise l'affinité **INTEGER** se comporte comme une colonne avec l'affinité **NUMERIC**. La différence entre les affinités **INTEGER** et **NUMERIC** ne devient évidente que dans les expressions de transtypage (**CAST expression**).

Un colonne avec l'affinité REAL se comporte comme une colonne avec l'affinité **NUMERIC** saur qu'elle force les valeurs entières à une représentation en point flottant. (Comme optimisation interne, les petites valeurs à point flottant sans composante fractionnaire et enregistrées dans des colonnes avec une affinité **REAL** sont écrits sur le disque comme des entiers dans le but de prendre moins d'espace et sont reconvertis en points flotttants lors de la lecture du disque. Cette optimisation est complètement transparente au niveau du SQL et peut être détectée en examinant les bits de la ligne du fichier de la base).

Une colonne avec l'affinité NONE ne préfère pas une classe d'enregistrement plutôt qu'une autre et ne cherche pas à forcer une donnée d'une classe d'enregistrement vers une autre.

2.1. Détermination de l'affinité d'une colonne

L'affinité d'une colonne est déterminé par le type déclaré dans la colonne, selon les règles suivantes prises dans l'ordre :

1. Si le type déclaré contient la chaîne "**INT**" alors c'est l'affinité **INTEGER** qui est assignée.
2. Si le type déclaré pour la colonnes contient soit "**CHAR**, soit "**CLOB**", soit "**TEXT**" alors la colonne possède l'affinité TEXT. Il faut noter que le type VARCHAR contient la chaîne "CHAR" et la colonne possède l'affinité TEXT.
3. Si le type déclaré pour une colonne contient la chaîne "**BLOB**" ou qu'aucun type n'est spécifié alors la colonne a l'affinité **NONE**.
4. Si le type de la colonne contient l'une des chaînes de 4 caractères "**REAL**", "**FLOA**" ou "**DOUB**" alors la colonne a l'affinité **REAL**.
5. Sinon, l'affinité est **NUMERIC**.

Noter que l'ordre des règles pour déterminer l'afffinité de la colonne est important. Une colonne dont le type déclaré est "**CHARINT**" satisfait à la fois les règles 1 et 2 mais la règle 1 jouit de précédence et ainsi la colonne a une affinité qui est **INTEGER.**

2.2. Exemples d'affinités déterminées par le nom

La table qui suit montre comment de noms de types de données des implémentations traditionnelles sont converties en affinités par les 5 règles de la section qui précède. Cette table montre seulement une petite partie des noms de types que **SQLite** peut accepter. Noter que l'argument numérique parenthésé qui suit le nom de type

(ex : "**VARCHAR(255)**") est ignoré par **SQLite** - **SQLite** n'impose aucune restriction de longueur (autre que la limite supérieure globale **SQLite_MAX_LENGTH**) sur la longueur des chaînes, **BLOB**s ou valeurs numériques.

Exemple de noms de type utilisés dans la commande CREATE TABLE ou l'expression CAST	affinité résultante	Règle utilisée pour définir l'affinité
INT **INTEGER** **TINYINT** **SMALLINT** **MEDIUMINT** **BIGINT** **UNSIGNED BIG INT** **INT2** **INT8**	**INTEGER**	**1**
CHARACTER(20) **VARCHAR(255)** **VARYING CHARACTER(255)** **NCHAR(55)** **NATIVE CHARACTER(70)** **NVARCHAR(100)** **TEXT** **CLOB**	**TEXT**	**2**
BLOB ***pas de type spécifié***	**NONE**	**3**
REAL **DOUBLE** **DOUBLE PRECISION** **FLOAT**	**REAL**	**4**
NUMERIC **DECIMAL(10,5)** **BOOLEAN** **DATE** **DATETIME**	**NUMERIC**	**5**

Noter qu'un type déclaré "**FLOATING POINT**" donne une affinité I**NTEGER**, et pas une affinité **REAL**, étant donnée la présence de "**INT**" à la fin de "**POINT**". Et le type déclaré **STRING** a une affinité **NUMERIC**, et non **TEXT,** selon la règle 5.

2.3. Exemple de comportement d'une affinité de colonne

```
CREATE TABLE maTable(
   t  TEXT,     -- affinité TEXT / règle 2
   nu NUMERIC,  -- affinité NUMERIC / règle 5
   i  INTEGER,  -- affinité INTEGER / règle 1
   r  REAL,     -- affinité REAL / règle 4
   no BLOB      -- affinité NONE / règle 3
);
```

```
-- enregistrement de valeurs TEXT, INTEGER, INTEGER, REAL, TEXT.
INSERT INTO maTable
   VALUES('500.0', '500.0', '500.0', '500.0', '500.0');
SELECT typeof(t), typeof(nu), typeof(i), typeof(r), typeof(no)
   FROM maTable;
>>> text|integer|integer|real|text

-- enregistrement de valeurs TEXT, INTEGER, INTEGER, REAL, REAL.
DELETE
   FROM maTable;
INSERT INTO maTable
   VALUES(500.0, 500.0, 500.0, 500.0, 500.0);
SELECT typeof(t), typeof(nu), typeof(i), typeof(r), typeof(no)
   FROM maTable;
>>> text|integer|integer|real|real

-- enregistrement de valeurs TEXT, INTEGER, INTEGER, REAL, INTEGER.
DELETE
   FROM t1;
INSERT INTO maTable
   VALUES(500, 500, 500, 500, 500);
SELECT typeof(t), typeof(nu), typeof(i), typeof(r), typeof(no)
FROM maTable;
>>> text|integer|integer|real|integer

-- les BLOBs sont toujours enregistrés comme BLOBs
-- indépendamment de l'affinité de la colonne.
DELETE
   FROM maTable;
INSERT INTO maTable
   VALUES(x'0500', x'0500', x'0500', x'0500', x'0500');
SELECT typeof(t), typeof(nu), typeof(i), typeof(r), typeof(no)
   FROM maTable;
>>> blob|blob|blob|blob|blob

-- les NULLs ne sont pas affectés par l'affinité
DELETE
   FROM maTable;
INSERT INTO maTable
   VALUES(NULL,NULL,NULL,NULL,NULL);
```

SELECT typeof(t), typeof(nu), typeof(i), typeof(r), typeof(no)

FROM maTable;

>>> null|null|null|null|null

3. Comparaisons

SQLite version 3 dispose de l'ensemble des opérateurs de comparaison du SQL, avec :

"=", "==", "<", "<=", ">", ">=", "!=", "<>",

"IN", "NOT IN", "BETWEEN", "IS", "IS NOT", .

3.1. Ordre de tri

Les résultats d'une comparaison dépendent des classes d'enregistrement des opérandes, selon les règles suivantes :

* Une valeur avec une classe d'enregistrement **NULL** est considérée comme inférieurs à n'importe quelle valeur (y compris une autre valeur dont la classe d'enregistrement est **NULL**).

* Une valeur **INTEGER** ou **REAL** est inférieure à une valeur **BLOB**. Quand un **INTEGER** ou un **REAL** est comparé à un autre **INTEGER** ou **REAL**, une comparaison numérique est réalisée.

* Une valeur **TEXT** est inférieure à une valeur **BLOB**. Quand deux valeurs **TEXT** sont comparées, une fonction de comparaison appropriée est réalisée pour déterminer le résultat.

* Quand deux valeurs **BLOB** sont comparées, le résultat est déterminé selon la fonction **`memcmp()`** (fonction de comparaison octet par octet de zones mémoire en C, C++, shell Linux).

3.2. Affinité des opérandes de comparaison

SQLite peut chercher à convertir des valeurs entre classes d'enregistrement **INTEGER**, **REAL** et/ou **TEXT** avant de réaliser une comparaison. Le fait qu'une conversion soit tentée ou non avant une comparaison dépend de l'affinité des opérandes. L'affinité des opérandes est déterminée par les règles qui suivent :

* Une expression qui est une simple référence à la valeur d'une colonne a la même affinité que la colonne. Noter que si X et Y.Z sont des noms de colonne, alors +X et +Y.Z sont les expressions prise en compte pour la façon de déterminer l'affinité.

* Une expression de la forme **`"CAST(`*`expr`*` AS `*`type`*`)"`** a une affinité qui est identique à celle d'une colonne de type déclaré **`"`*`type`*`"`**

* Sinon, une expression a une affinité **NONE**.

3.3. Priorité des conversions de type pour une comparaison

"***Appliquer une affinité***" signifie convertir un opérande en une classe d'enregistrement particulière si et seulement si la conversion est sans perte et réversible. L'affinité est appliquée aux opérandes d'un opérateur de comparaison selon les règle de priorité qui suivent, à prendre dans l'ordre :

* Si un opérande a l'affinité **INTEGER**, **REAL** ou **NUMERIC** et que l'autre opérande a une affinité **TEXT** ou **NONE**, alors l'affinité **NUMERIC** est appliquée à l'autre opérande.

* Si un opérande a l'affinité **TEXT** et l'autre opérande une affinité **NONE**, alors l'affinité **TEXT** est appliquée à l'autre opérande.

* Sinon, aucune affinité n'est appliquée et les deux opérandes sont comparés tels quels.

L'expression **`"a BETWEEN b AND c"`** est traitée comme deux comparaisons séparées :

`"a >= b AND a <=c"` même si cela implique que des affinités différentes soient appliquées à **`'a'`** dans chacune des comparaisons. Les conversions de types de données dans des comparaisons de la forme **`"x in (SELECT y …)"`** sont conduites comme si la comparaison était en réalité **`"x=y"`**. L'expression **`"a IN(x, y, z, …)"`** est équivalente à :**`"a = +x OR a = +y OR a = +z OR …".`** En d'autres termes, les valeurs à droite de l'opérateur **`IN (`**les valeurs **`"x", "y"`** et **`"z")`** dans l'exemple) sont considérées comme n'ayant aucune affinité, même si il leur arrive d'être des valeurs de colonne ou d'expressions **CAST.**

3.4. Exemple de comparaison

CREATE TABLE maTable(

a TEXT, -- affinité TEXT

```
   b NUMERIC,   -- affinité NUMERIC
   c BLOB,      -- aucune affinité
   d            -- aucune affinité
);

-- Les valeurs seront enregistrées comme TEXT, INTEGER, TEXT,
-- et INTEGER respectivement
INSERT INTO maTable
   VALUES('500', '500', '500', 500);
SELECT typeof(a), typeof(b), typeof(c), typeof(d)
   FROM maTable;
>>> text|integer|text|integer

-- La colonne "a" a une affinité TEXT, alors les valeurs
-- numériques du côté droit de la comparaison
-- sont converties en TEXT avant la comparaisons

SELECT a < 40, a < 60, a < 600
   FROM maTable;
>>> 0|1|1

-- L'affinité TEXT est appliquée à la partie droite de l'opérande
-- mais quand ils sont déjà TEXT aucune conversion ne se produit
SELECT a < '40', a < '60', a < '600'
   FROM maTable;
>>> 0|1|1

-- La colonne "b" a une affinité NUMERIC  et cette affinité est
-- imposée à l'opérande de drroite. Quand l'opérande est déjà NUMERIC
-- imposer cette affinité n'a aucun effet.
-- Toutes les valeurs sont comparées numériquement.
SELECT b < 40, b < 60, b < 600
   FROM maTable;
>>> 0|0|1

-- L'affinité Numeric est appliquée aux opérandes de droite,
-- les convertissant de textes en entiers
-- La comparaison est numérique
SELECT b < '40', b < '60', b < '600'
   FROM maTable;
```

```
>>> 0|0|1

-- Aucune conversion d'affinité n'a lieu. Les valeurs du côté droit
-- ont toute la classe d'enregistrement INTEGER qui est toujours
-- inférieure à la valeur TEXT de droite.
SELECT c < 40, c < 60, c < 600
   FROM maTable;
>>> 0|0|0

-- Aucune conversion : les valeurs son comparées à TEXT.
SELECT c < '40', c < '60', c < '600'
   FROM maTable;
>>> 0|1|1

-- Aucune conversion : les valeurs du côté droit on pour classe
-- d'enregistrement INTEGER et sont comparées aux valeurs INTEGER
-- du côté gauche.
SELECT d < 40,   d < 60,   d < 600
   FROM maTable;
>>> 0|0|1

-- Aucune conversion d'affinité n'a lieu. les valeurs INTEGER
-- à gauche sont inférieures aux valeurs TEXT values à droite.
SELECT d < '40', d < '60', d < '600'
   FROM maTable;
>>> 1|1|1
```

Toutes les comparaisons de ces exemples peuvent être commutées, par exemple **`"a<40"`** peut être réécrit **`"40>a"`** : le résultat est le même.

4. Opérateurs

Tous les opérateurs mathématiques (**`+, -, *, /, %, <<, >>, &,`** et **`|`**) adaptent les opérandes des classes d'enregistrement avant de s'appliquer. L'adaptation est mise en œuvre même si elle provoque des pertes et qu'elle n'est pas réversible. Un opérande **NULL** dans un opérateur mathématique provoque un résultat **NULL**. Un opérande sur un opérateur mathématique qui n'est pas vu comme numérique ni **NULL** est converti en **`0`** ou **`0.0`**.

5. SELECT trié, groupé et composé

Quand les résultats d'une requête sont triés par une clause **ORDER BY**, Le valeurs de classe d'enregistrement **NULL** viennent en premier, suivies des valeurs **INTEGER** et **REAL** classés selon la valeur numérique. Suivent ensuite les valeurs **TEXT** dans l'ordre lexicographique, et enfin les valeurs **BLOB** dans l'ordre donné par **`memcmp()`**.

Pour les valeurs groupées par la clause **GROUP BY**, les valeurs ayant des classes d'enregistrement différentes sont considérées comme distinctes, sauf pour les valeurs **INTEGER** et **REAL** qui sont considérées comme égales si numériquement, elles sont égales. Aucune affinité n'est appliquée à une valeur de résultat d'un

groupement issu de la clause **GROUP.**

Les opérateurs **SELECT** composés avec **UNION**, **INTERSECT** et **EXCEPT** réalisent implicitement les comparaisons entre valeurs. Aucune affinité n'est appliquée aus opérandes comparés dans les comparaisons implicites associées à **UNION**, **INTERSECT** et **EXCEPT** - les valeurs sont comparées en l'état.

6. Fonctions de comparaison.

Quand **SQLite** compare deux chaînes, il utilise une *séquence de comparaison* ou une *fonction de comparaison* (c'est la même chose) pour déterminer quelle chaîne est la plus grande ou si les deux chaînes son égales. **SQLite** a trois fonctions internes de : **BINARY**, **NOCASE** et **RTRIM.**

* **BINARY** - Compare les données chaîne en utilisant **`memcmp()`**, sans se soucier de l'encodage.

* **NOCASE** - même chose que **BINARY** sauf que les 26 caractères ASCII de haute casse (majuscules) sont remplacés par leur équivalent de basse casse (minuscule) avant que l* comparaison ne soit engagée. Noter que les seul caractères ASCII sont concernés. **SQLite** ne cherche pas à considérer la casse UTF complète en raison de la taille des tables qui seraient requises.

* **RTRIM** - Même chose que BINAY, sauf que les espaces en queue de la chaîne sont ignorés.

Une application peut enregistrer des fonctions de comparaisons additionnelle en utilisant l'interface **`SQLite3_create_collations()`**.

6.1. Assignation d'une fonction de comparaison en SQL

Chaque colonne de chaque table a une fonction de comparaison associée. Si aucune fonction n'est explicitement définie, alors la fonction de comparaison par défaut est **BINARY**. La clause **COLLATE** de la **définition de colonne** est utilisée pour définir des fonctions alternatives de comparaison pour la colonne.

La règle pour déterminer quelle fonction de comparaison doit être utilisée pour une opérateur de comparaisons binaire (**`=, <, >, <=,>=, !=, IS,`** et **`IS NOT`**) se détermine comme suit, en prenant les règles dans l'ordre :

1. Si chaque opérande a une fonction de comparaison assignée en utilisant la contrainte postfixée **COLLATE** ***opérateur***, alors la fonction de comparaison explicite est utilisée, avec précédence de la fonction de comparaison de l'opérande gauche.

2. Si chaque opérande est une colonne, alors la fonction de comparaison de cette colonne est utilisée avec précédence de la fonction de comparaison de l'opérande gauche. À propos de ce qui vient d'être dit, un nom de colonne précédé d'un ou plusieurs opérateurs unaires **`"+"`** est encore considéré comme un nom de colonne.

3. Sinon, la fonction de comparaisons BINARY est utilisée pour la comparaison.

Un opérande de comparaison est considéré comme ayant une fonction de comparaison explicite (règle 1 ci-dessus) si une sous expression au moins de l'opérande n'utilise la contrainte postfixée **COLLATE** ***opérateur.*** Ainsi si un **COLLATE** ***opérateur*** est utilisé n'importe où dans une expression de comparaison, la fonction de comparaison définie par cet opérateur est utilisé pour la comparaison de chaînes dès que les colonnes de tables peuvent constituer une partie de cette expression. S'il y a deux sous expressions **COLLATE** ***opérateur*** ou davantage qui apparaissent n'importe où dans une comparaison, la plus à gauche est utilisée, indépendamment de la profondeur d'insertion de **Collate** ***opérateur*** dans l'expression et indépendamment du parenthésage.

L'expression "**`x BETWEEN y AND z`**" est logiquement équivalente à une double comparaison **`"x >= y AND x <= z"`**et fonctionne en respectant les fonctions de comparaison comme si on avait deux comparaisons séparées. L'expression **`"x IN (SELECT y …)"`** est conduite de la même manière que si c'était l'expression "x = y" en ce qui concerne la fonction de comparaison. La fonction de comparaison **`"x IN (y, z …)"`** est la fonction de comparaison de "x".

Enfin, la clause **ORDER BY** qui est incluse dans une commande **SELECT** peut posséder une fonction de comparaison utilisant un **COLLATE** ***opérateur***, auquel cas la fonction de comparaison est utilisée pour le tri. Ainsi, si l'expression triée par une clause **ORDER BY** est une colonne, alors la fonction de comparaison de la colonnes est utilisée pour déterminer l'ordre du tri. Si l'expression ,n'est pas une colonne et a une clause **COLLATE**, alors la fonction de comparaison **BINARY** est utilisée.

6.2. Exemples de fonction de comparaison

Les exemples qui suivent identifient la fonction de comparaison qui doit être uilisée pour déterminer les résultats de comparaisons de textes qui peuvent apparaître dans diverses commandes SQL. Noter qu'une comparaison de texte peut ne pas être requise, ni aucune fonction de comparaison, dans le cas de valeurs **NUMERIC**, **BLOB** ou **NULL**.

```
CREATE TABLE maTable(
   x INTEGER PRIMARY KEY,
   a,              /* fonction de comparaison BINARY */
   b COLLATE BINARY,  /* fonction de comparaison BINARY */
   c COLLATE RTRIM,   /* fonction de comparaison RTRIM  */
   d COLLATE NOCASE   /* fonction de comparaison NOCASE */
);
/* x   a     b     c       d */
INSERT INTO maTable VALUES(1,'abc','abc', 'abc  ','abc');
INSERT INTO maTable VALUES(2,'abc','abc', 'abc',  'ABC');
INSERT INTO maTable VALUES(3,'abc','abc', 'abc ', 'Abc');
INSERT INTO maTable VALUES(4,'abc','abc ','ABC',  'abc');

/* La comparaison a=b est réalisée avec la fonction de
  comparaison BINARY  */
SELECT x
   FROM maTable
   WHERE a = b ORDER BY x;
>>> 1 2 3

/* La comparaison de textes a=b utilise la fonction de
  comparaison RTRIM */
SELECT x
   FROM maTable
   WHERE a = b COLLATE RTRIM ORDER BY x;
>>> 1 2 3 4

/* La comparaison de textes d=a utilise la fonction de
  comparaison NOCASE */
SELECT x
   FROM maTable
   WHERE d = a ORDER BY x;
>>> 1 2 3 4

/* La comparaison a=d est réalisée avec la fonction de
  comparaison BINARY  */
SELECT x
   FROM maTable
   WHERE a = d ORDER BY x;
>>> 1 4
```

```
/* La comparaison de textes 'abc'=c utilise la fonction de
  comparaison RTRIM */
SELECT x
   FROM maTable
   WHERE 'abc' = c ORDER BY x;
>>> 1 2 3

/* La comparaison de textes c='abc' utilise la fonction de
  comparaison RTRIM */
SELECT x
   FROM maTable
   WHERE c = 'abc' ORDER BY x;
>>> 1 2 3

/* le groupement est réalisé avec la fonction de comparaison
   NOCASE (les valeurs 'abc', 'ABC', et 'Abc' sont placées
   dans le même groupe */
SELECT count(*)
   FROM maTable GROUP BY d ORDER BY 1;
>>> 4

/* Le groupement est réalisé en utilisant la fonction de comparaison  the BINARY.  'abc' et 'ABC'
et 'Abc' forment différents groupes */
SELECT count(*)
   FROM maTable GROUP BY (d || '') ORDER BY 1;
>>> 1 1 2

/* Tri de la colonne c utilisant la fonction de comparaison RTRIM */
SELECT x
   FROM maTable ORDER BY c, x;
>>> 4 1 2 3

/* Tri de (c||'') réalisé avec la fonction de comparaison BINARY */
SELECT x
   FROM maTable ORDER BY (c||''), x;
>>> 4 2 3 1

/* Tri de la colonne c utilisant la fonction de comparaison NOCASE */
SELECT x
   FROM maTable ORDER BY c COLLATE NOCASE, x;
```

>>> 2 4 3 1