Cours de Programmation Déclarative et Bases de Données

SQLite

Nicolas Jouandeau

n@up8.edu

2022

particularités de SQLite

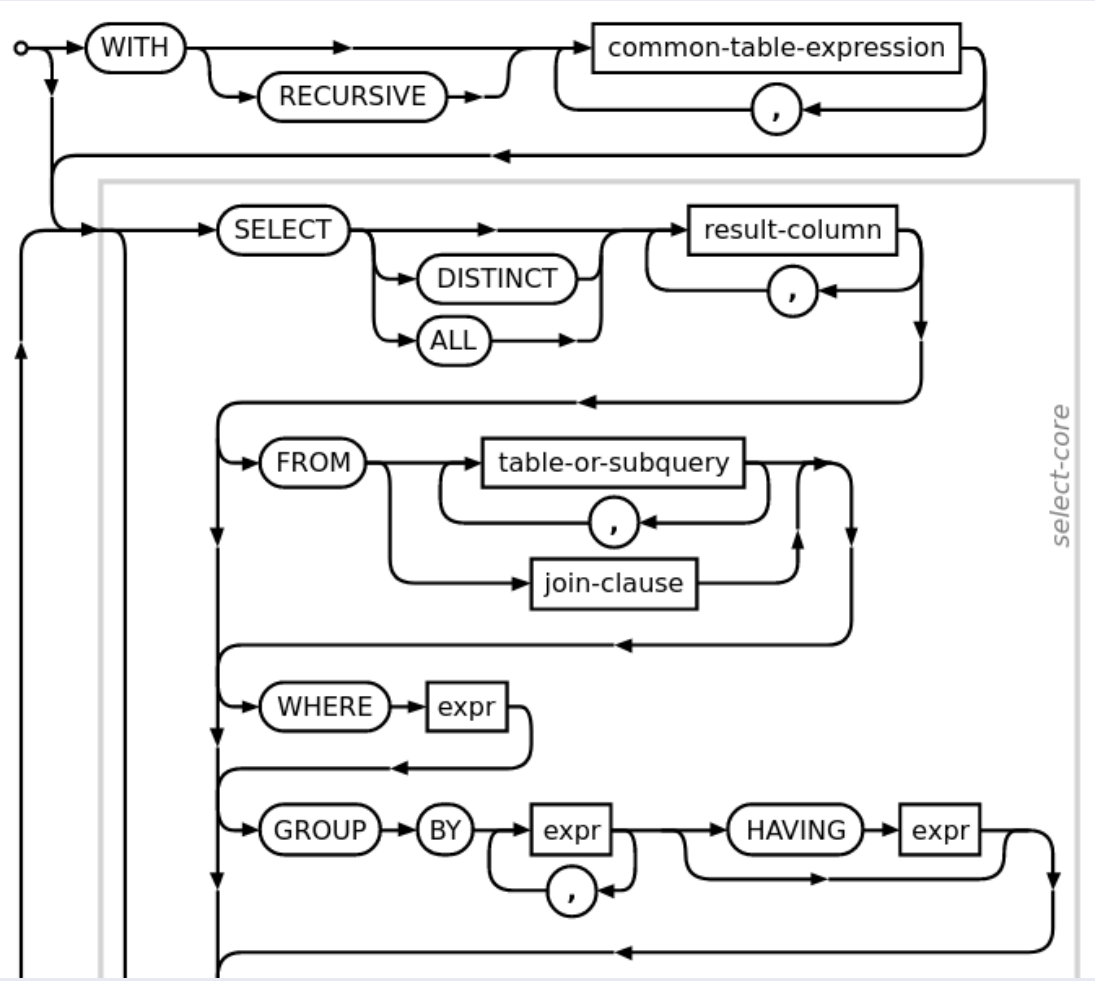
- utilisable dans les smartphones et périphériques portables
- développé en C-ANSI (i.e. 2 fichiers, sqlite3.c et sqlite3.h)
- pas de configuration et pas de service (≠ MySQL)
- assure les propriétés ACID
- typage dynamique dans les tables
  (i.e. indépendance des types des champs)
- accès simultané à plusieurs bases de données
- création de bases de données en RAM pour un accès plus rapide
- `https://www.sqlite.org`

selon la documentation SQLite

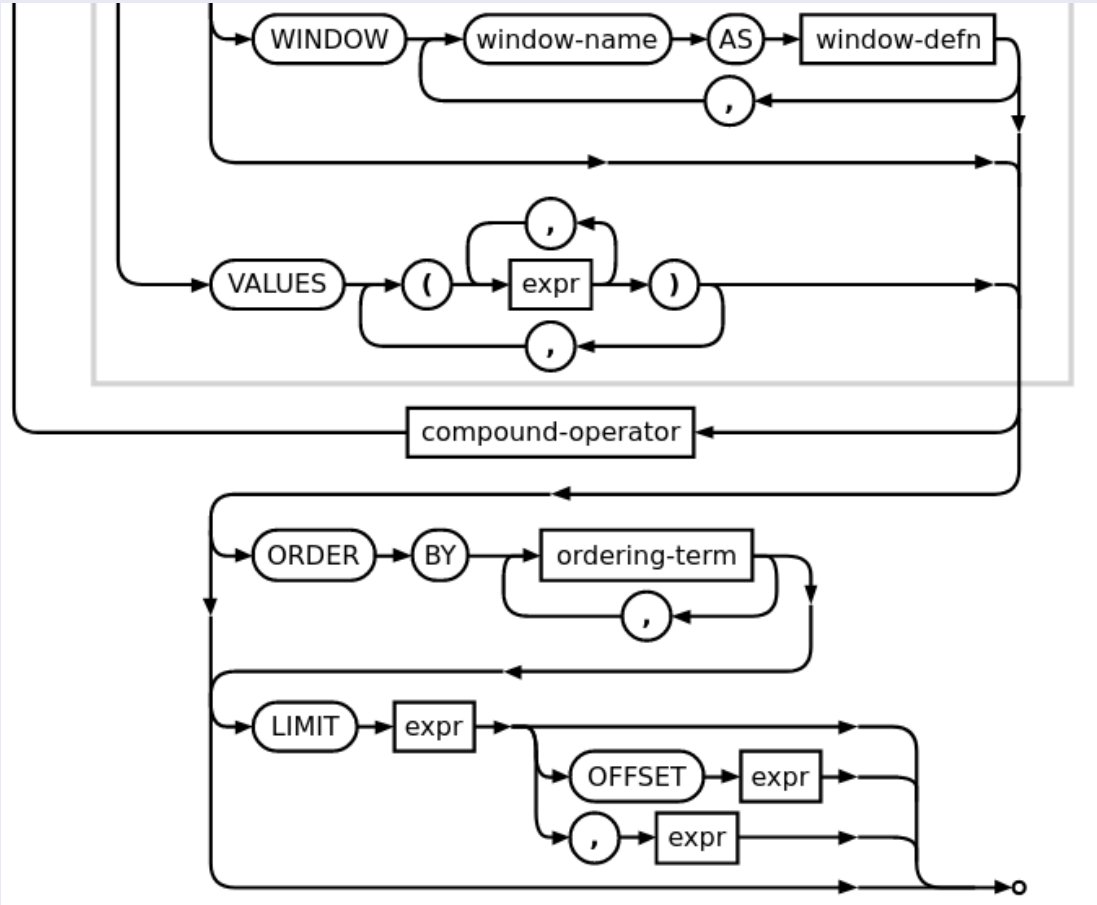
- un fichier .db
- moins de 1GO par BLOB (Binary Large OBject)
- moins de 2000 colonnes par table
- moins de 1Go de résultat par requète

diagrammes et commandes

- `https://sqlite.org/lang.html`
- exemple du `SELECT`



diagrammes et commandes (suite)



SQLite en pratique

- installation sur Ubuntu : `sudo apt-get install sqlite3`
- les commandes SQLite sont précédées d'un "."

```
$> sqlite3
 SQLite version 3.31.1 2020-01-27 19:55:54
 Enter ".help" for usage hints.
 Connected to a transient in-memory database.
 Use ".open FILENAME" to reopen on a persistent database.

sqlite> .quit
$>
```

- le raccourci Ctrl-D est équivalent à la commande `.quit`

bases de données SQLite

- une base de données SQLite
  - correspond à un fichier
  - est créée dans le répertoire courant
  - possède une extension ".db"
- créer une nouvelle base nommée A

```
$> sqlite3 A.db
```

- utiliser une base nommée A

```
$> sqlite3 A.db
```

- supprimer la base nommée A

```
$> rm -f A.db
```

commandes de mise en forme

```
sqlite> .headers on       # afficher les en-tetes des resultats
sqlite> .header off       # par défaut
sqlite> .mode column      # afficher en colonne
sqlite> .mode list        # par défaut
sqlite> .separator ", "   # définir le séparateur inter-colonne
sqlite> .width 10 10 10   # définir la largeur des colonnes
sqlite> .mode html        # pour un affichage en mode HTML
sqlite> .show             # afficher les paramètres d'affichage

sqlite> .show
        echo: off
         eqp: off
     explain: auto
     headers: off
        mode: list
   nullvalue: ""
      output: stdout
colseparator: "|"
rowseparator: "\n"
       stats: off
       width:
    filename: :memory:
```

base de données et table

- utiliser une base de données

```
$> sqlite3 People.db
```

- changer de base de données

```
sqlite> .open People.db
sqlite> .open People
```

- créer une table

```
sqlite> CREATE TABLE Enseignant (id INTEGER PRIMARY KEY, \
  name VARCHAR(10), last_name VARCHAR(10));
```

- les types internes pour le stockage des données :
  - NULL
  - INTEGER : valeur signée entière, sur 1 à 8 octets
  - REAL : valeur flottante sur 8 octets
  - NUMERIC : valeur numérique
  - TEXT : chaine de caractères encodés UTF-8, UTF-16BE ou UTF-16LE
  - BLOB : données binaires

types de données

- dans une base de données classique, le type d'une donnée est définie par le type du champ dans lequel la donnée est placée
- avec SQLite, le type des données n'est pas défini par le champ dans lequel la donnée est placée
- type interne = type défini dans la requète d'ajout de la donnée
- affinité = concordance entre type de la donnée et type interne
- table d'affinité ordonnée avec les types des SGBD
  1. INTEGER = INT, TINYINT, SMALLINT, MEDIUMINT, BIGINT, UNSIGNED BIG INT, INT2, INT8
  2. TEXT = CHARACTER(20), VARCHAR(255), VARYING CHARACTER(255), NCHAR(55), NATIVE CHARACTER(70), NVARCHAR(100), CLOB
  3. BLOB = si pas de type
  4. REAL = DOUBLE, DOUBLE PRECISION, FLOAT
  5. NUMERIC = DECIMAL(10,5), BOOLEAN, DATE, DATETIME
- affinité également définie pour les expressions

format et contraintes

- plusieurs formats de date
  - un TEXT au format ISO8601 ("YYYY-MM-DD HH:MM:SS.SSS")
  - un REAL comptant le nombre de jour passé depuis un jour dit zéro
  - un INTEGER tel que les systèmes Unix compte le temps

    soit le nombre de secondes depuis le 1er janvier 1970 (heure UTC)
- contraintes sur les champs
  - NOT NULL : ne peut pas être null
  - AUTOINCREMENT : est incrémenté automatiquement
  - CHECK : vérifie une expresion
    (par exemple pour a, on vérifie "CHECK(length(a) >= 10)")

    si la contrainte n'est pas vérifié, le résultat de la requête se solde par un échec

```
sqlite> CREATE TABLE all_tel ( id INTEGER PRIMARY KEY AUTOINCREMENT,
                               tel TEXT NOT NULL CHECK(length(tel) >= 10));
sqlite> INSERT INTO all_tel VALUES(0,'123');
 Error: CHECK constraint failed: all_tel
sqlite> INSERT INTO all_tel VALUES(0,'0123456789');
sqlite>
```

format et contraintes (suite)

- contraintes sur les champs (suite)
  - DEFAULT : précise une valeur par défaut
  - COLLATE : précise une méthode de comparaison des string (BINARY, NOCASE ou RTRIM)
    - BINARY : comportement identique à memcmp
    - NOCASE : sans différencier majuscule et minuscule
    - RTRIM : BINARY en ignorant les espaces de fin
  - PRIMARY KEY : quand un ou plusieurs champs constitue une clé primaire
  - UNIQUE : quand une valeur doit être unique
  - FOREIGN KEY : quand c'est une clé secondaire (i.e. clé venant d'une autre table)

```
sqlite> CREATE TABLE groups (group_id INTEGER PRIMARY KEY, name TEXT NOT NULL);
sqlite> CREATE TABLE pt (pt_id INTEGER PRIMARY KEY, name TEXT NOT NULL, \
  group_id, FOREIGN KEY (group_id) \
  REFERENCES groups (group_id) );
```

gestion des tables

- lister les tables existantes

```
sqlite> .tables
```

- lister les commandes utilisées pour créer les tables existantes ou pour une table particuliere

```
sqlite> .schema
 CREATE TABLE Enseignant (id INTEGER PRIMARY KEY, name VARCHAR(10), \
  last_name VARCHAR(10));
 CREATE TABLE Blabla (id INTEGER PRIMARY KEY, what TEXT);

sqlite> .schema Enseignant
 CREATE TABLE Enseignant (id INTEGER PRIMARY KEY, name VARCHAR(10), \
  last_name VARCHAR(10));
```

- lister les bases existantes/chargées/liées

```
sqlite> .databases
 main: <ABSOLUTE_PATH>/People.db
```

- créer, renommer puis supprimer une table

```
sqlite> CREATE TABLE Blabla (id INTEGER PRIMARY KEY, what TEXT);
sqlite> ALTER TABLE Blabla RENAME TO Blablabla;
sqlite> DROP TABLE Blablabla;
```

- ajouter un enregistrement dans une table

```
sqlite> INSERT INTO Enseignant VALUES(1, 'Nicolas', 'J');
```

gestion des tables (suite)

- faire un SELECT

```
sqlite> SELECT * FROM Enseignant;
 1|Nicolas|J

sqlite > SELECT * FROM Enseignant WHERE id=1;
 1|Nicolas|J
```

- supprimer un enregistrement

```
sqlite> INSERT INTO Enseignant VALUES(2, 'Another', 'E');

sqlite> SELECT * FROM Enseignant;
 1|Nicolas|J
 2|Another|E

sqlite> DELETE FROM Enseignant where id=2;

sqlite> SELECT * FROM Enseignant;
 1|Nicolas|J
```

- ajouter une colonne

```
sqlite> ALTER TABLE Enseignant ADD COLUMN "bureau";

sqlite> ALTER TABLE Enseignant ADD COLUMN "cours1";

sqlite> SELECT * FROM Enseignant;
 1|Nicolas|J||
```

gestion des tables (suite)

- MAJ une valeur dans une table

```
sqlite> UPDATE Enseignant SET bureau=A193 WHERE id=1;

sqlite> SELECT * FROM Enseignant;
 1|Nicolas|J|A193|
```

- enregistrer les résultats dans un fichier

```
sqlite> .output output.txt

sqlite> SELECT * FROM Enseignant;

sqlite> .quit

$> more output.txt
  1|Nicolas|J|A193|
```

- dumper une table au format SQL dans un fichier

```
sqlite> .output People.sql

sqlite> .dump People
```

- dump par commande shell

```
$> sqlite3 People.db .dump > People.sql
```

gestion des tables (suite)

- lire les commandes d'un fichier (dump ou autre)

```
sqlite> .read People.sql
```

- lire la sortie d'un programme comme des commandes

```
sqlite> .read '|prg.sh'
```

- lecture par redirection dans un shell

```
$> sqlite3 People.db < People.sql
```

```
$> sqlite3 People.db | prg.sh
```

- redirection du résultat d'une requête dans un fichier

```
$> sqlite3 database "SELECT * FROM table;" >  somefile
```

```
$> sqlite3 database "SELECT * FROM table;" >> somefile
```

jointure en SQLite

- ⚠ les jointures RIGHT et FULL ne sont pas supportées en SQLite

```
sqlite> SELECT * FROM A JOIN B;
sqlite> SELECT * FROM A JOIN B WHERE A.id=1;
sqlite> SELECT * FROM A JOIN B WHERE A.id=1 AND B.id=1;
sqlite> SELECT * FROM A LEFT JOIN B WHERE A.id=1 AND B.id=1;
sqlite> SELECT * FROM A CROSS JOIN B WHERE A.id=1 AND B.id=1;
sqlite> SELECT * FROM A FULL OUTER JOIN B WHERE A.id=1 AND B.id=1;
```

- quand on recherche des valeurs de A n'ayant pas de correspondance dans B

```
SELECT ... FROM A LEFT JOIN B WHERE ...
```

- quand on recherche des valeurs de B n'ayant pas de correspondance dans A

```
SELECT ... FROM B LEFT JOIN A WHERE ...
```

- pour obtenir le produit cartésien (chq ligne de A combinée avec chq ligne de B)

```
SELECT ... FROM A CROSS JOIN B WHERE ...
```

- pour obtenir l'union des valeurs de deux tables, en complètant les valeurs incomplètes

```
SELECT ... FROM A FULL OUTER JOIN B WHERE ...
```

- pour obtenir des tests sur plusieurs références d'une même table (exemple les champs c1 associés à différents champs c2)

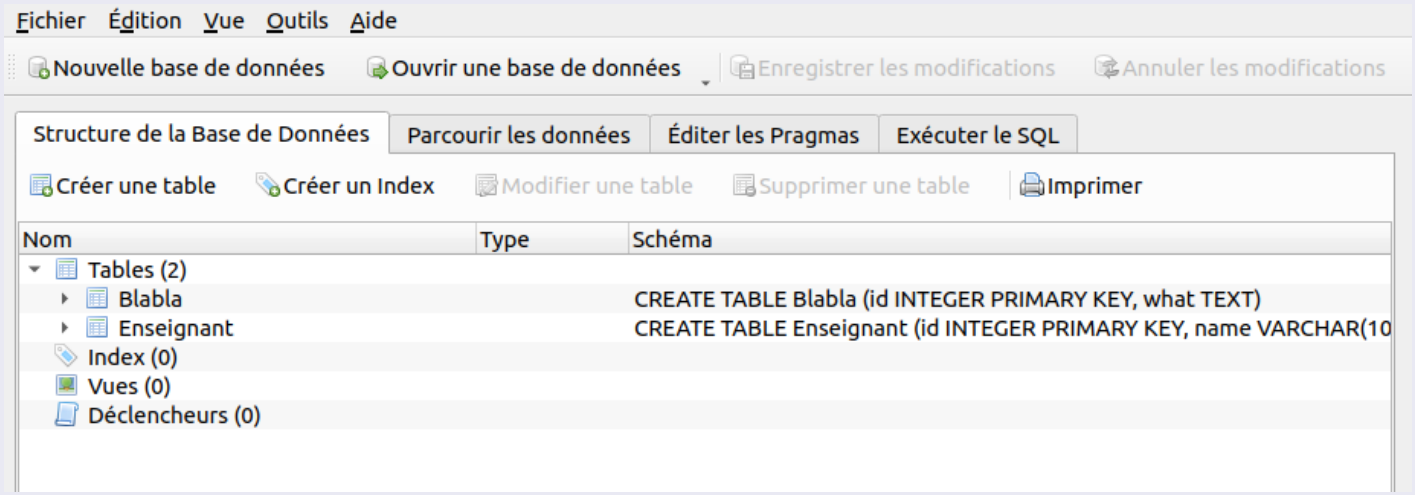
```
SELECT ... FROM A a INNER JOIN A b ON a.id=b.id AND a.c1 <> b.c2 ...
```

interface sqlitebrowser

- interface graphique SQLite pour browser

```
$> sqlite3 People.db
 sqlite> CREATE TABLE Enseignant (id INTEGER PRIMARY KEY, name VARCHAR(10), \
    last_name VARCHAR(10));
 sqlite> CREATE TABLE Blabla (id INTEGER PRIMARY KEY, what TEXT);
 sqlite> INSERT INTO Enseignant VALUES(1, 'Nicolas', 'J');
 sqlite> INSERT INTO Blabla VALUES(1, 'blablabla...');
 sqlite> .quit
$>

$> sqlitebrowser
```



accès via un programme Python3

- `https://docs.python.org/3/library/sqlite3.html`

Principe

- `connect()` pour se connecter à la base de données
- `cursor()` pour récupérer un `cursor`
- `execute()` pour exécuter une requête
- `close()` sur le `cursor`
- `close()` sur la connexion

Assurer la sauvegarde des modifications

- `conn.commit()` avant le `close()` de l'objet de connexion

Recupérer les éléments du `cursor`

- `cur.fetchone()` pour récupérer un élément
  (un élément est un tuple)
- `cur.fetchall()` pour récupérer tous les éléments
  (les éléments sont une liste de tuples)

SELECT * avec select-sqlite.py

```
#!/usr/bin/env python
import sqlite3
try :
  conn = sqlite3.connect('People.db')
  c = conn.cursor()
  c.execute("SELECT * FROM Enseignant")
  res = c.fetchall()
  for i in res:
    print(i)
  c.close()
  conn.close()
except:
  print("SELECT failed")
```

exécution

```
$> python3 select-sqlite.py
 (1, 'Nicolas', 'J')
 (2, 'Another', 'E')
$>
```

SELECT avec WHERE avec select2-where-sqlite.py

```
#!/usr/bin/env python
import sqlite3
try :
  conn = sqlite3.connect('People.db')
  c = conn.cursor()
  req = "SELECT * FROM Enseignant WHERE last_name = ?"
  c.execute(req, ("J",))
  res = c.fetchone()      # !!! on récupère le premier élément
  c.close()
  conn.close()
  print(res)
except:
  print("SELECT failed")
```

exécution

```
$> python3 select2-where-sqlite.py
 (1, 'Nicolas', 'J')
$>
```

SELECT avec WHERE avec select3-where-sqlite.py

```
#!/usr/bin/env python
import sqlite3
try :
  conn = sqlite3.connect('People.db')
  c = conn.cursor()
  req = "SELECT * FROM Enseignant WHERE last_name = ?"
  c.execute(req, ("J",))
  res = c.fetchall()        # !!! on récupère tous les éléments
  c.close()
  conn.close()
  print(res)
except:
  print("SELECT failed")
```

exécution

```
$> python3 select3-where-sqlite.py
 [(1, 'Nicolas', 'J')]
$>
```

ajouter un enregistrement avec `add-sqlite.py`

```
#!/usr/bin/env python
import sqlite3
try :
  conn = sqlite3.connect('People.db')
  c = conn.cursor()
  req = "INSERT INTO Enseignant VALUES (?,?,?)"
  c.execute(req, (11,"Nicolas","A"))
  conn.commit()
  c.close()
  conn.close()
except:
  print("INSERT failed")
```

exécution

```
$> python3 add-sqlite.py
$>
```

retirer un enregistrement avec `del-sqlite.py`

```
#!/usr/bin/env python
import sqlite3
try :
  conn = sqlite3.connect('People.db')
  c = conn.cursor()
  req = "DELETE FROM Enseignant WHERE id = ?"
  c.execute(req, (11,))
  conn.commit()
  c.close()
  conn.close()
except:
  print("DELETE failed")
```

exécution

```
$> python3 del-sqlite.py
$>
```

accès via un programme Racket

- https://docs.racket-lang.org/db/index.html
- https://docs.racket-lang.org/db/query-api.html

sqlite-with-racket.rkt

```
#lang racket
(require db)
(define C (sqlite3-connect #:database "/home/n/People.db"))
(query C "SELECT * FROM Enseignant")
```

exécution dans un terminal

```
$> racket sqlite-with-racket.rkt
(rows-result
 '(((name . "id") (decltype . "INTEGER"))
   ((name . "name") (decltype . "VARCHAR(10)"))
   ((name . "last_name")
    (decltype . "VARCHAR(10)")))
 '(#(1 "Nicolas" "J") ))
```

autres SGBD accessibles en Racket

- postgresql : relationnelle open source
- mysql : relationnelle format Oracle
- cassandra : distribuée NoSQL
- odbc : relationnelle format Microsoft

types de requêtes en Racket

- `query` : pour récupérer des résultats avec l'entête de typage
- `query-exec` : pour une requête qui n'attend pas de résultat
- `query-rows` : pour une requête avec pour résultat une liste de vector(s)
  chq enregistrement est placé dans un vector
- `query-list` : pour une requête avec pour résultat une liste de valeurs
- `query-row` : pour une requête avec pour résultat une ligne de champs
  les champs spécifiés dans la requête sont placés dans le vector résultat
- `query-maybe-row` : pour une requête avec pour résultat une ou zéro ligne
  si le résultat est zéro ligne, la fonction retourne #f
- `query-value` : pour une requête avec pour résultat une valeur
- `query-maybe-value` : pour une requête avec pour résultat une ou zéro valeur

fonctions utiles

- `(list-tables c)` : liste de string contenant les noms des tables
- `(table-exists? c TABLE-NAME)` : prédicat pour l'existance d'une table

exemples de requêtes en Racket

```
#lang racket
(require db)
(define C (sqlite3-connect #:database "/home/n/People.db"))
(query C "SELECT name FROM sqlite_master")
(list-tables C)
(table-exists? C "Blabla")
(query-rows C "SELECT name FROM sqlite_master")
(query C "SELECT * FROM Enseignant")
(query-rows C "SELECT * FROM Enseignant")
(query-list C "SELECT name FROM Enseignant")
(query-row C "SELECT * FROM Enseignant WHERE id=1")

(query-rows C "SELECT * FROM Enseignant WHERE id = $1" 1)
(query-rows C "SELECT * FROM Enseignant WHERE name > ?" "B")
(query-exec C "INSERT INTO Enseignant VALUES (3, \"Another\", \"F\")")

(query-exec C "CREATE TABLE Cours (n INTEGER, d VARCHAR(20))")
(query-exec C "DROP TABLE Cours")
(query-rows C "SELECT * FROM Cours")
(query-exec C "INSERT INTO Cours values (0, 'Racket')")
(disconnect C)
```

accès via un programme C

- `https://www.sqlite.org/quickstart.html`
- `https://www.sqlite.org/capi3ref.html`
- installation : `$> sudo apt-get install -y libsqlite3-dev`
- compilation avec l'option "`-l sqlite3`"

principales fonctions

- `sqlite3()` : pour se connecter à une base de données
- `sqlite3_prepare()` et `sqlite3_finalize()` : pour initier et clore une requète
- `sqlite3_open()` et `sqlite3_close()` : pour ouvrir et fermer une base
- `sqlite3_bind()` : définir un format pour les entiers et les alphanumériques dans les requêtes
- `sqlite3_step()` : passer à l'enregistrement suivant
- `sqlite3_column()` : récupérer les colonnes de l'enregistrement courant
- `sqlite3_exec()` : exécuter une requète

sqlite-with-c.c (début)

```
#include <stdio.h>
#include <sqlite3.h>
static int callback(void *data, int argc, char **argv, char **azColName);
// gcc sqlite-with-c.c -l sqlite3
int main(int argc, char* argv[]) {
   sqlite3* db;
   char* zErrMsg = 0;
   int rc = sqlite3_open("People.db", &db);
   if( rc ) {
      fprintf(stderr, "Can't open database: %s\n", sqlite3_errmsg(db));
      return(0);
   } else {
      fprintf(stderr, "Opened database successfully\n");
   }
   char* sql = "SELECT * from Enseignant";
   const char* data = "Callback function called";
   rc = sqlite3_exec(db, sql, callback, (void*)data, &zErrMsg);
   if( rc != SQLITE_OK ) {
      fprintf(stderr, "SQL error: %s\n", zErrMsg);
      sqlite3_free(zErrMsg);
   } else {
      fprintf(stdout, "Operation done successfully\n");
   }
   sqlite3_close(db);
}
```

sqlite-with-c.c (fin)

```
static int callback(void *data, int argc, char **argv, char **azColName){
  printf("%s: \n", (const char*)data);
   for(int i = 0; i<argc; i++)
      printf("%s = %s ", azColName[i], argv[i] ? argv[i] : "NULL");
   printf("\n");
   return 0;
}
```

exécution

```
$> ./a.out
Opened database successfully
Callback function called:
id = 1 name = Nicolas last_name = J
Operation done successfully
$>
```