L'interface ligne de commande de SQLite

Ce fichier est une traduction de la page html pour la documentation officielle de l'interface en ligne de commande de SQLite : `https://www.sqlite.org/cli.html`. Le site est régulièrement mis à jour, la version traduite date de juin 2024.

Table of Contents

1 Pour commencer

Le projet SQLite fournit un programme de ligne de commande simple `sqlite3` (ou `sqlite3.exe` sous Windows) qui permet à l'utilisateur de saisir et d'exécuter des instructions SQL sur une base de données SQLite ou sur une archive ZIP. Ce document fournit une introduction sur la façon d'utiliser le programme sqlite3 .

Pour commencer, lancez le programme sqlite3 en tapant `sqlite3` dans un shell, éventuellement suivi du nom du fichier contenant la base de données SQLite(ou l'archive ZIP). Si le fichier nommé n'existe pas, un nouveau fichier de base de données portant le nom donné sera créé automatiquement. Si aucun fichier de base de données n'est spécifié sur la ligne de commande, une base de données temporaire est créée et automatiquement supprimée en fin d'exécution du programme `sqlite3`.

Au démarrage, le programme `sqlite3` affichera un bref message de bannière puis vous invitera à saisir du SQL. Tapez les instructions SQL (terminées par un point-virgule), appuyez sur "Entrée" et le code SQL sera exécuté.

Par exemple, pour créer une nouvelle base de données SQLite nommée `ex1` avec une seule table nommée `tbl1`, vous pouvez procéder comme suit :

```
$ sqlite3 ex1
SQLite version 3.36.0 2021-06-18 18:36:39
Enter ".help" for usage hints.
sqlite> create table tbl1(one varchar(10), two smallint);
sqlite> insert into tbl1 values('hello!',10);
sqlite> insert into tbl1 values('goodbye', 20);
sqlite> select * from tbl1;
hello!|10
goodbye|20
sqlite>
```

Terminez le programme sqlite3 en tapant le caractère de fin de fichier de votre système (généralement un `CTRL`-D). Utilisez le caractère d'interruption (généralement un `CTRL`-C) pour arrêter une instruction SQL qui durt trop longtemps.

Assurez-vous de taper un point-virgule à la fin de chaque commande SQL ! Le programme sqlite3 recherche un point-virgule pour savoir si l'instruction SQL est terminée. Si vous omettez le point-virgule, `sqlite3` affiche une invite de suite et attend que vous saisissiez plus de texte pour terminer l'instruction SQL en court. Cette fonctionnalité vous permet de saisir des commandes SQL qui s'étendent sur plusieurs lignes. Par exemple:

```
sqlite> CREATE TABLE tbl2 (
   ...>   f1 varchar(30) primary key,
   ...>   f2 text,
   ...>   f3 real
   ...> );
sqlite>
```

2 Démarrage en deux clics sous Windows

Les utilisateurs de Windows peuvent double-cliquer sur l'icône `sqlite3.exe` pour que le shell de ligne de commande affiche une fenêtre de terminal exécutant SQLite. Cependant, comme un double-clic démarre `sqlite3.exe` sans arguments de ligne de commande, aucun fichier de base de données n'aura été spécifié, donc SQLite utilisera une base de données temporaire qui sera supprimée à la fin de la session. Pour utiliser un fichier de disque persistant comme base de données, entrez la commande `.open` immédiatement après le démarrage de la fenêtre du terminal :

```
SQLite version 3.36.0 2021-06-18 18:36:39
Enter ".help" for usage hints.in
Connected to a transient in-memory database.
Use ".open FILENAME" to reopen on a persistent database.
sqlite> .open ex1.db
sqlite>
```

L'exemple ci-dessus provoque l'ouverture et l'utilisation du fichier de base de données nommé `ex1.db`. Le fichier "ex1.db" est créé s'il n'existe pas auparavant. Vous souhaiterez peut-être utiliser un chemin d'accès complet pour vous assurer que le fichier se trouve dans le répertoire dans lequel vous pensez qu'il se trouve. Utilisez des barres obliques comme caractère de séparation de répertoire. En d'autres termes, utilisez `c:/work/ex1.db`, et non `c:\work\ex1.db`.

Alternativement, vous pouvez créer une nouvelle base de données en utilisant le stockage temporaire par défaut, puis enregistrer cette base de données dans un fichier disque à l'aide de la commande `.save` :

```
SQLite version 3.36.0 2021-06-18 18:36:39
Enter ".help" for usage hints.
Connected to a transient in-memory database.
Use ".open FILENAME" to reopen on a persistent database.
sqlite> ... ici on entre plusieurs commandes SQL  ...
sqlite> ... en oublient d'ouvrir une base  ...
sqlite> .save ex1.db
sqlite>
```

Soyez prudent lorsque vous utilisez la commande `.save` car elle écrasera tous les fichiers de base de données préexistants portant le même nom sans demander de confirmation. Comme pour la commande `.open`, vous souhaiterez peut-être utiliser un chemin d'accès complet avec des séparateurs de répertoire par barre oblique pour éviter toute ambiguïté.

3 Les commandes point de sqlite3

La plupart du temps, `sqlite3` lit simplement les lignes d'entrée et les transmet à la bibliothèque SQLite pour exécution. Mais les lignes d'entrée commençant par un point (`.`) sont interceptées et interprétées par le programme `sqlite3` lui-même. Ces *commandes point* sont généralement utilisées pour modifier le format de sortie des requêtes ou pour exécuter certaines instructions de requête préemballées. À l'origine, il n'y avait que quelques commandes par points, mais au fil des années, de nombreuses nouvelles fonctionnalités se sont accumulées, si bien qu'il en existe aujourd'hui plus de 60.

Pour obtenir une liste des commandes point disponibles, vous pouvez saisir `.help` sans argument. Ou entrez `.help TOPIC` pour des informations détaillées sur `TOPIC`.

Voici la liste de ces options, avec l'explication traduite en français.

`.archive ...`	Gérer les archives SQL
`.auth ON|OFF`	Afficher les rappels de l'autorisateur
`.backup ?DB? FILE`	Sauvegarder `DB` (par défaut "main") dans `FILE`
`.bail on|off`	Arrêter après une erreur. Par défaut `OFF`
`.binary on|off`	Activer ou désactiver la sortie binaire. Par défaut `OFF`
`.cd DIRECTORY`	Changer le répertoire de travail pour `DIRECTORY`
`.changes on|off`	Afficher le nombre de lignes modifiées
`.check GLOB`	Échouer si la sortie depuis `.testcase` ne correspond pas
`.clone NEWDB`	Clôner les données dans `NEWDB` à partir de la base de données existante
`.databases`	Lister les noms et fichiers des bases de données attachées
`.dbconfig ?op? ?val?`	Lister ou changer les options de `sqlite3_db_config()`
`.dbinfo ?DB?`	Afficher les informations de statut sur la base de données
`.dump ?TABLE? ...`	Traduire tout le contenu de la base de données en instruction SQL
`.echo on|off`	Activer ou désactiver l'écho des commandes
`.eqp on|off|full`	Activer ou désactiver la plan `EXPLAIN QUERY PLAN`

`.excel`	Afficher la sortie de la prochaine commande dans une feuille de calcul
`.exit ?CODE?`	Quitter ce programme avec le code de retour `CODE`
`.expert`	(EXPÉRIMENTAL) Suggérer des index pour les requêtes spécifiées
`.fullschema ?--indent?`	Afficher le schéma et le contenu des tables `sqlite_stat`
`.headers on|off`	Activer ou désactiver l'affichage des en-têtes
`.help ?-all? ?PATTERN?`	Afficher le texte d'aide pour `PATTERN`
`.import FILE TABLE`	Importer les données de `FILE` dans `TABLE`
`.imposter INDEX TABLE`	Créer une `TABLE .imposter` sur l'index `INDEX`
`.indexes ?TABLE?`	Afficher les noms des index
`.iotrace FILE`	Activer la journalisation de diagnostic d'E/S dans FILE
`.limit ?LIMIT? ?VAL?`	Afficher ou changer la valeur d'un `SQLITE_LIMIT`
`.lint OPTIONS`	Signaler les problèmes potentiels du schéma
`.load FILE ?ENTRY?`	Charger une bibliothèque d'extensions
`.log FILE|off`	Activer ou désactiver la journalisation. FILE peut être stderr/stdout
`.mode MODE ?TABLE?`	Définir le mode de sortie
`.nullvalue STRING`	Utiliser STRING à la place des valeurs `NULL`
`.once (-e|-x|FILE)`	Sortie de la prochaine commande SQL uniquement dans `FILE`
`.open ?OPTIONS? ?FILE?`	Fermer la base de données existante et rouvrir `FILE`
`.output ?FILE?`	Envoyer la sortie vers `FILE` ou stdout si `FILE` est omis
`.print STRING...`	Imprimer la chaîne de caractères littérale `STRING`
`.prompt MAIN CONTINUE`	Remplacer les invites standard

`.quit`	Quitter ce programme
`.read FILE`	Lire l'entrée de `FILE`
`.restore ?DB? FILE`	Restaurer le contenu de `DB` (par défaut "main") à partir de `FILE`
`.save FILE`	Écrire la base de données en mémoire dans `FILE`
`.scanstats on|off`	Activer ou désactiver les métriques `sqlite3_stmt_scanstatus()`
`stats ?on/?off`	Montre les statistiques ou les arrête
`.schema ?PATTERN?`	Afficher les déclarations CREATE correspondant à `PATTERN`
`.selftest ?OPTIONS?`	Exécuter les tests définis dans la table `SELFTEST`
`.separator COL ?ROW?`	Changer les séparateurs de colonne et de ligne
`.session ?NAME? CMD ...`	Créer ou contrôler des sessions
`.sha3sum ...`	Calculer un hachage SHA3 du contenu de la base de données
`.shell CMD ARGS...`	Exécuter `CMD ARGS...` dans un shell système
`.show`	Afficher les valeurs actuelles pour divers paramètres
`.stats ?on|off?`	Afficher les statistiques ou activer/désactiver les statistiques
`.system CMD ARGS...`	Exécuter `CMD ARGS...` dans un environnement de shell
`.tables ?TABLE?`	Lister les noms des tables correspondant au modèle `LIKE TABLE`
`.testcase NAME`	Commencer à rediriger la sortie vers `testcase-out.txt`
`.timeout MS`	Essayer d'ouvrir les tables verrouillées pendant `MS` millisecondes
`.timer on|off`	Activer ou désactiver le chronomètre SQL
`.trace FILE|off`	Afficher chaque instruction SQL au fur et à mesure de son exécution

`.vfsinfo ?AUX?`	Informations sur le Système de fichiers Virtuels (VFS) de premier niveau
`.vfslist`	Lister tous les Systèmes de fichiers Virtuels (VFS) disponibles
`.vfsname ?AUX?`	Imprimer le nom de la pile du Systèmes de fichiers Virtuels (VFS)
`.width NUM1 NUM2 ...`	Définir les largeurs de colonne pour le mode `column`

4 Règles pour les commandes point

4.1 Structure d'une ligne

L'entrée de la ligne de commande d'interface est analysée en une séquence composée de :

- Instructions SQL ;
- commandes point;
- Commentaires de la ligne de commande d'interface.

Les instructions SQL ont des formes libres et peuvent être réparties sur plusieurs lignes, avec des espaces ou des commentaires SQL intégrés n'importe où. Ils se terminent soit par un `;` caractère à la fin d'une ligne de saisie, ou un caractère `/` ou le mot `go` sur une ligne seul. Lorsqu'il n'est pas à la fin d'une ligne de saisie, le `;` le caractère agit pour séparer les instructions SQL. Les espaces de fin sont ignorés à des fins de terminaison.

Une commande point a une structure plus restrictive :

- Il doit commencer par son `.` dans la marge de gauche sans espace précédent.
- Il doit être entièrement contenu sur une seule ligne d'entrée.
- Cela ne peut pas se produire au milieu d'une instruction SQL ordinaire. En d'autres termes, cela ne peut pas se produire à une invite de continuation.
- Il n'y a pas de syntaxe de commentaire pour les commandes point.

L'interface en ligne de commande accepte également les commentaires sur une ligne entière commençant par un caractère `#` et s'étendant jusqu'à la fin de la ligne. Il ne faut aucun espace avant le caractère `#`.

4.2 Arguments des commandes point

Les arguments passés aux commandes point sont analysés en commençant par la queue de la commande en suivant ces règles :

1. La nouvelle ligne de fin et tout autre espace de fin sont supprimés ;
2. Les espaces immédiatement après le nom de la commande point ou toute limite de fin d'entrée d'argument sont ignorés ;
3. Une entrée d'argument commence par n'importe quel caractère autre qu'un espace ;
4. Une entrée d'argument se termine par un caractère qui dépend ainsi de son caractère principal :
   - pour un guillemet simple de début (`'`), un guillemet simple fait office de délimiteur de fin ;
   - pour un guillemet double (`"`), un guillemet double non échappé fait office de délimiteur de fin ;
   - pour tout autre caractère principal, le délimiteur de fin est n'importe quel espace ;
   - la commande `tail` end agit comme délimiteur de fin pour tout argument ;
   - Dans une entrée d'argument entre guillemets doubles, un guillemet double avec une barre oblique inverse fait partie de l'argument plutôt que de son guillemet final ;

- Dans un argument entre guillemets doubles, la traduction traditionnelle de la séquence d'échappement de la chaîne C littérale et de la barre oblique inverse est effectuée ;
- Les délimiteurs d'entrée d'argument (les guillemets ou les espaces) sont ignorés pour produire l'argument passé.

4.3 Exécution des commandes point

Les commandes point sont interprétées par le programme de ligne de commande sqlite3.exe, et non par SQLite lui-même. Ainsi, aucune des commandes point ne fonctionnera comme argument pour les interfaces SQLite telles que `sqlite3_prepare()` ou `sqlite3_exec()`.

5 Modification des formats de sortie

Le programme `sqlite3` est capable d'afficher les résultats d'une requête dans 14 formats de sortie différents :

- `ascii`
- `box`
- `csv`
- `column`
- `html`
- `insert`
- `json`
- `line`
- `list`
- `markdown`
- `quote`
- `table`
- `tabs`
- `tcl`

Vous pouvez utiliser la commande point `.mode` pour basculer entre ces formats de sortie. Le mode de sortie par défaut est `list`. En mode `list`, chaque ligne d'un résultat de requête est écrite sur une ligne de sortie et chaque colonne de cette ligne est séparée par une chaîne de séparation spécifique. Le séparateur par défaut est un symbole de barre verticale (`|`). Le mode liste est particulièrement utile lorsque vous souhaitez envoyer le résultat d'une requête à un autre programme (tel qu'AWK) pour un traitement supplémentaire.

```
sqlite> .mode list
sqlite> select * from tbl1 ;
bonjour!|10
au revoir|20
sqlite>
```

Tapez `.mode` sans argument pour afficher le mode actuel :

```
sqlite> .mode
mode de sortie actuel : liste
sqlite>
```

Utilisez la commande point `.separator` pour changer le séparateur. Par exemple, pour remplacer le séparateur par une virgule et un espace, vous pouvez procéder comme suit :

```
sqlite> .separator ", "
sqlite> select * from tbl1 ;
bonjour!, 10
au revoir, 20
sqlite>
```

La prochaine commande `.mode` pourrait réinitialiser le `.separator` à une valeur par défaut (en fonction de ses arguments). Vous devrez donc probablement répéter la commande

`.separator` chaque fois que vous changez de mode si vous souhaitez continuer à utiliser un séparateur non standard.

En mode `quote`, la sortie est formatée sous forme d'instruction SQL littérale. Les chaînes sont placées entre guillemets simples et les guillemets simples internes sont échappés en les doublant. Les blobs sont affichés en notation littérale hexadécimale (Ex : x'abcd'). Les nombres sont affichés sous forme de texte ASCII et les valeurs `NULL` sont affichées sous forme de `NULL`. Toutes les colonnes sont séparées les unes des autres par une virgule (ou tout autre caractère sélectionné à l'aide de `.separator`).

```
sqlite> .mode quote
sqlite> select * from tbl1 ;
'bonjour !',10
'au revoir',20
sqlite>
```

En mode `line`, chaque colonne d'une ligne de la base de données est affichée seule sur une ligne. Chaque ligne comprend le nom de la colonne, un signe égal et les données de la colonne. Les enregistrements successifs sont séparés par une ligne vierge. Voici un exemple de sortie en mode `line` :

```
sqlite> .mode line
sqlite> select * from tbl1 ;
un = bonjour !
deux = 10

un = au revoir
deux = 20
sqlite>
```

En mode colonne, chaque enregistrement est affiché sur une ligne distincte avec les données alignées en colonnes. Par exemple:

```
sqlite> colonne .mode
sqlite> select * from tbl1 ;
un deux
--------- ---
Bonjour! dix
au revoir 20
sqlite>
```

En mode `column` (et également en modes `box`, `table` et `markdown`) la largeur des colonnes s'ajuste automatiquement. Mais vous pouvez remplacer cela en fournissant une largeur spécifiée pour chaque colonne à l'aide de la commande `.width`. Les arguments de `.width` sont des entiers qui correspondent au nombre de caractères à consacrer à chaque colonne. Les nombres négatifs signifient justifier à droite.

Ainsi :

```
sqlite> .width 12 -6
sqlite> select * from tbl1 ;
un deux
------------ ------
Bonjour! dix
```

```
au revoir 20
sqlite>
```

Une largeur de 0 signifie que la largeur de la colonne est choisie automatiquement. Les largeurs de colonnes non spécifiées deviennent nulles. Par conséquent, la commande `.width` sans argument réinitialise toutes les largeurs de colonnes à zéro et provoque donc la détermination automatique de toutes les largeurs de colonnes.

Le mode « colonne » est un format de sortie tabulaire. Les autres formats de sortie tabulaires sont « box », « markdown » et « table » :

```
sqlite> .width
sqlite> .mode markdown
sqlite> select * from tbl1 ;
|    un     | deux |
|---------- |----- |
| Bonjour!  |  10  |
| au revoir |  20  |
sqlite> .mode table
sqlite> select * from tbl1 ;
+-----------+------+
|    un     | deux |
+-----------+------+
| Bonjour!  |  10  |
| au revoir |  20  |
+-----------+------+
sqlite> .mode box
sqlite> select * from tbl1 ;
+-----------+-----+
|    un     |deux |
+-----------+-----+
| bonjour ! | 10  |
| au revoir | 20  |
+-----------+-----+
sqlite>
```

Les modes en colonnes ont quelques options supplémentaires pour contrôler le formatage. L'option `--wrap` *N* (où *N* est un entier) amène les colonnes à envelopper le texte qui dépasse N caractères. Le wrapping est désactivé si N est nul.

sqlite> insert into tbl1 values('Le renard saute par-dessus le chien.',90); sqlite> .mode box –wrap 30 sqlite> select * from tbl1 où deux>50 ; +——————————+——+ | un |deux | +——————————+——+ | Le renard saute par-dessus le chien. | 90 | +——————————+——+ sqlite>

Le retour à la ligne se produit après exactement *N* caractères, qui peuvent se trouver au milieu d'un mot. Pour revenir à la limite d'un mot, ajoutez l'option `--wordwrap on` (ou simplement `-ww` pour faire court) :

```
sqlite> .mode box --wrap 30 -ww
sqlite> select * from tbl1 où deux>50 ;
+-----------------------------------+------+
```

```
|             un                         | deux |
+----------------------------------------+------+
| Le renard saute par-dessus le chien. |  90  |
+----------------------------------------+------+
sqlite>
```

L'option `--quote` fait que les résultats de chaque colonne sont cités comme une chaîne SQL littérale, comme dans le mode `quote`. Consultez l'aide en ligne pour des options supplémentaires.

La commande `.mode box --wrap 60 --quote` est si utile pour les requêtes de base de données à usage général qu'elle reçoit son propre alias. Au lieu de taper toute cette commande de 27 caractères, vous pouvez simplement dire `.mode qbox`.

Un autre mode de sortie utile est `insertion`. En mode insertion, la sortie est formatée pour ressembler aux instructions `SQL INSERT`. Utilisez le mode insertion pour générer du texte qui pourra ensuite être utilisé pour saisir des données dans une autre base de données.

Lors de la spécification du mode d'insertion, vous devez donner un argument supplémentaire qui est le nom de la table dans laquelle insérer. Par exemple:

```
sqlite> .mode insert new_table
sqlite> select * from tbl1 où deux<50 ;
INSERT INTO "new_table" VALUES('bonjour',10);
INSERT INTO "new_table" VALUES('goodbye',20);
sqlite>
```

Les autres modes de sortie incluent `csv`, `json` et `tcl`. Essayez-les vous-même pour voir ce qu'ils font.

6 Requête sur le schéma de la base de données

Le programme `sqlite3` fournit plusieurs commandes pratiques et utiles pour consulter le schéma de la base de données. Il n'y a rien de ce que font ces commandes qui ne puisse être fait par d'autres moyens. Ces commandes sont fournies uniquement à titre de raccourci.

Par exemple, pour voir une liste des tables de la base de données, vous pouvez saisir `.tables`.

```
sqlite> .tables
tbl1 tbl2
sqlite>
```

La commande `.tables` est similaire à la définition du mode `list` puis à l'exécution de la requête suivante :

```
SELECT name FROM sqlite_master
WHERE type IN ('table','view') AND name NOT LIKE 'sqlite_%'
ORDER BY 1
```

Mais la commande `.tables` fait plus. Il interroge la table `sqlite_master` pour toutes les bases de données attachées, pas seulement la base de données principale. Et il organise sa sortie en colonnes soignées.

La commande `.indexes` fonctionne de la même manière pour lister tous les index. Si la commande `.indexes` reçoit un argument qui est le nom d'une table, alors elle affiche uniquement les index de cette table.

La commande `.schema` affiche le schéma complet de la base de données, ou pour une seule table si un argument facultatif de nom de table est fourni :

```
sqlite> .schéma
créer la table tbl1 (un varchar (10), deux smallint)
CRÉER UNE TABLE tbl2 (
  clé primaire f1 varchar(30),
  texte f2,
  f3 réel
);
sqlite> .schéma tbl2
CRÉER UNE TABLE tbl2 (
  clé primaire f1 varchar(30),
  texte f2,
  f3 réel
);
sqlite>
```

La commande `.schema` est à peu près la même que pour définir le mode liste, puis saisir la requête suivante :

```
SELECT sql FROM sqlite_schema
ORDER BY tbl_name, tapez DESC, nom
```

Comme pour `.tables`, la commande `.schema` affiche le schéma de toutes les bases de données attchées. Si vous souhaitez uniquement voir le schéma d'une seule base de données (peut-être `main`), vous pouvez ajouter un argument à `.schema` pour restreindre sa sortie :

```
sqlite> .schema principal.*
```

La commande `.schema` peut être complétée par l'option `--indent`, auquel cas elle tente de reformater les différentes instructions CREATE du schéma afin qu'elles soient plus facilement lisibles par les humains.

La commande `.databases` affiche une liste de toutes les bases de données ouvertes dans la connexion actuelle. Il y en aura toujours au moins deux. La première est la *principale*, la base de données d'origine ouverte. La seconde est appelée *temp*, la base de données utilisée pour les tables temporaires. Des bases de données supplémentaires peuvent être répertoriées pour les bases de données attachées à l'aide de l'instruction `ATTACH`. La première colonne de sortie est le nom avec lequel la base de données est attachée et la deuxième colonne de résultat est le nom du fichier externe. Il peut y avoir une troisième colonne de résultat qui sera soit `r/o` soit `r/w` selon que le fichier de base de données est en lecture seule ou en lecture-écriture. Et il pourrait y avoir une quatrième colonne de résultats affichant le résultat de `sqlite3_txn_state()` pour ce fichier de base de données.

```
sqlite> .databases
```

La commande point `.fullschema` fonctionne comme la commande `.schema` dans la mesure où elle affiche l'intégralité du schéma de la base de données. Mais `.fullschema` inclut également les dumps des tables de statistiques `sqlite_stat1`, `sqlite_stat3` et `sqlite_stat4`, si elles existent. La commande `.fullschema` fournit normalement toutes les informations nécessaires pour recréer exactement un plan de requête pour une requête spécifique. Lorsqu'ils signalent des problèmes suspectés avec le planificateur de requêtes SQLite à l'équipe de développement SQLite, les développeurs sont priés de fournir la sortie `.fullschema` complète dans le cadre du rapport de problème. Notez que les tables `sqlite_stat3` et `sqlite_stat4` contiennent des échantillons d'entrées d'index et peuvent donc contenir des données sensibles. N'envoyez donc pas la sortie `.fullschema` d'une base de données propriétaire sur un canal public.

7 Ouverture des fichiers de base de données

La commande `.open` ouvre une nouvelle connexion à la base de données, après avoir d'abord fermé la commande de base de données précédemment ouverte. Dans sa forme la plus simple, la commande `.open` invoque simplement `sqlite3_open()`

sur le fichier nommé comme argument. Utilisez le nom `:memory:` pour ouvrir une nouvelle base de données en mémoire qui disparaît lorsque la CLI se ferme ou lorsque la commande `.open` est à nouveau exécutée. Ou n'utilisez aucun nom pour ouvrir une base de données privée et temporaire sur disque qui disparaîtra également à la sortie ou à l'utilisation de `.open`.

Si l'option `--new` est incluse avec `.open`, alors la base de données est réinitialisée avant d'être ouverte. Toutes les données antérieures sont détruites. Il s'agit d'un écrasement détruisant les données antérieures et aucune confirmation n'est demandée, utilisez donc cette option avec précaution.

L'option `--readonly` ouvre la base de données en mode lecture seule. L'écriture sera interdite.

L'option `--deserialize` entraîne la lecture de l'intégralité du contenu du fichier sur disque en mémoire, puis son ouverture en tant que base de données en mémoire à l'aide de l'interface `sqlite3_deserialize()`. Bien entendu, cela nécessitera beaucoup de mémoire si vous disposez d'une grande base de données. De plus, toutes les modifications que vous apportez à la base de données ne seront pas enregistrées sur le disque, sauf si vous les enregistrez explicitement à l'aide des commandes `.save` ou `.backup`.

L'option `--append` entraîne l'ajout de la base de données SQLite à un fichier existant plutôt que de fonctionner comme un fichier autonome. Voir l'extension `appendvfs` pour plus d'informations.

L'option `--zip` fait que le fichier d'entrée spécifié est interprété comme une archive `ZIP` plutôt que comme un fichier de base de données SQLite.

L'option `--hexdb` permet de lire le contenu de la base de données à partir des lignes d'entrée suivantes au format hexadécimal, plutôt qu'à partir d'un fichier séparé sur le disque. L'outil de ligne de commande `dbtotxt` peut être utilisé pour générer le texte approprié pour une base de données. L'option `--hexdb` est destinée à être utilisée par les développeurs SQLite à des fins de test. Nous ne connaissons aucun cas d'utilisation de cette option en dehors des tests et du développement internes de SQLite.

8 Redirection des E/S

8.1 Écrire les résultats dans un fichier

Par défaut, `sqlite3` envoie les résultats de la requête vers la sortie standard. Vous pouvez modifier cela à l'aide des commandes `.output` et `.once`. Mettez simplement le nom d'un fichier de sortie comme argument dans `.output` et tous les résultats des requêtes ultérieures seront écrits dans ce fichier. Ou utilisez la commande `.once` au lieu de `.output` et la sortie ne sera redirigée que pour que la commande suivante et ensuite les autres reviendront à la console. Utilisez `.output` sans argument pour recommencer à écrire sur la sortie standard. Par exemple:

```
sqlite> .mode list
sqlite> .separator |
sqlite> .output test_file_1.txt
sqlite> select * from tbl1 ;
sqlite> .exit
$ cat test_file_1.txt
bonjour|10
au revoir|20
$
```

Si le premier caractère du nom de fichier `.output` ou `.once` est le symbole de barre verticale (`|`), alors les caractères restants sont traités comme une commande et la sortie est envoyée à cette commande. Cela facilite le transfert des résultats d'une requête vers un autre processus. Par exemple, la commande `open -f` sur un Mac ouvre un éditeur de texte pour afficher le contenu qu'il lit à partir de l'entrée standard. Ainsi, pour voir les résultats d'une requête dans un éditeur de texte, on pourrait taper :

```
sqlite> .une fois | open -f
sqlite> SELECT * FROM bigTable;
```

Si les commandes `.output` ou `.once` ont un argument `-e`, alors la sortie est collectée dans un fichier temporaire et l'éditeur de texte système est invoqué sur ce fichier texte. Ainsi, la commande `.once -e` obtient le même résultat que `.once '|open -f'` mais avec l'avantage d'être portable sur tous les systèmes.

Si les commandes `.output` ou `.once` ont un argument `-x`, ce qui les amène à accumuler la sortie sous forme de valeurs séparées par des virgules (CSV) dans un fichier temporaire, invoquez l'utilitaire système par défaut pour afficher les fichiers CSV. (généralement un tableur) sur le résultat. Il s'agit d'un moyen rapide d'envoyer le résultat d'une requête vers une feuille de calcul pour une visualisation facile :

```
sqlite> .once -x
sqlite> SELECT * FROM bigTable;
```

La commande `.excel` est un alias pour `.once -x`. Cela fait exactement la même chose.

8.2 Lire SQL à partir d'un fichier

En mode interactif, sqlite3 lit le texte saisi (soit des instructions SQL, soit des commandes point) à partir du clavier. Vous pouvez également rediriger les entrées d'un fichier lorsque

vous lancez sqlite3, bien sûr, mais vous n'avez alors pas la possibilité d'interagir avec le programme. Parfois, il est utile d'exécuter un script SQL contenu dans un fichier en saisissant d'autres commandes à partir de la ligne de commande. Pour cela, la commande point `.read` est fournie.

La commande `.read` prend un seul argument qui est (généralement) le nom d'un fichier à partir duquel lire le texte d'entrée.

```
sqlite> .read monscript.sql
```

La commande `.read` arrête temporairement la lecture à partir du clavier et prend à la place son entrée dans le fichier nommé. Une fois la fin du fichier atteinte, la saisie revient au clavier. Le fichier de script peut contenir des commandes point, tout comme une entrée interactive ordinaire.

Si l'argument de `.read` commence par le signe `|` , puis au lieu d'ouvrir l'argument sous forme de fichier, il exécute l'argument (sans le `|`) en tant que commande, puis utilise la sortie de cette commande comme entrée. Ainsi, si vous disposez d'un script qui génère du SQL, vous pouvez exécuter ce SQL directement à l'aide d'une commande similaire à la suivante :

```
sqlite> .read |monscript.bat
```

8.3 Fonctions d'Entrée/Sortie sur les fichiers

Le shell de ligne de commande ajoute deux fonctions SQL définies par l'application qui facilitent respectivement la lecture du contenu d'un fichier dans une colonne de table et l'écriture du contenu d'une colonne dans un fichier.

La fonction SQL `readfile(X)` lit tout le contenu du fichier nommé `X` et renvoie ce contenu sous forme d'un *BLOB*. Cela peut être utilisé pour charger du contenu dans un tableau. Par exemple:

```
sqlite> CREATE TABLE images(name TEXT, type TEXT, img BLOB);
sqlite> INSERT INTO images(name,type,img)
   ...>   VALUES('icon','jpeg',readfile('icon.jpg'));
```

La fonction SQL `writefile(X,Y)` écrit le *blob* `Y` dans le fichier nommé `X` et renvoie le nombre d'octets écrits. Utilisez cette fonction pour extraire le contenu d'une seule colonne de table dans un fichier. Par exemple:

```
sqlite> SELECT writefile('icon.jpg',img) FROM images WHERE name='icon';
```

Notez que les fonctions `readfile(X)` et `writefile(X,Y)` sont des extensions qui ne sont pas intégrées à la bibliothèque SQLite principale. Ces routines sont disponibles sous forme d'extensions chargeables dans le fichier source `ext/misc/fileio.c` qui fait partie du dépôt source de SQLite.

8.4 La fonction SQL edit()

L'interface en ligne de commande possède une autre fonction SQL intégrée nommée `edit()`. Cette fonction `edit()` prend un ou deux arguments. Le premier argument est une valeur – souvent une grande chaîne multiligne à modifier. Le deuxième argument est l'invocation d'un éditeur de texte. (Il peut inclure des options pour affecter le comportement de l'éditeur). Si le deuxième argument est omis, la variable d'environnement `VISUAL` est

utilisée. La fonction `edit()` écrit son premier argument dans un fichier temporaire, appelle l'éditeur sur le fichier temporaire, relit le fichier en mémoire une fois l'éditeur terminé, puis renvoie le texte modifié.

La fonction `edit()` peut être utilisée pour apporter des modifications à de grandes valeurs de texte. Par exemple:

```
sqlite> UPDATE docs SET body=edit(body) WHERE name='report-15';
```

Dans cet exemple, le contenu du champ `docs.body` pour l'entrée où `docs.name` est `report-15` sera envoyé à l'éditeur. Après le retour de l'éditeur, le résultat sera réécrit dans le champ `docs.body`.

L'opération par défaut de la commande `edit()` consiste à appeler un éditeur de texte. Mais en utilisant un programme d'édition alternatif dans le deuxième argument, vous pouvez également lui faire éditer des images ou d'autres ressources non textuelles. Par exemple, si vous souhaitez modifier une image JPEG qui se trouve être stockée dans un champ d'une table, vous pouvez exécuter :

```
sqlite> UPDATE pics SET img=edit(img,'gimp') WHERE id='pic-1542';
```

Le programme d'édition peut également être utilisé comme visualiseur, en ignorant simplement la valeur de retour. Par exemple, pour simplement regarder l'image ci-dessus, vous pouvez exécuter :

```
sqlite> SELECT length(edit(img,'gimp')) WHERE id='pic-1542';
```

8.5 Importation de fichiers au format CSV ou autres formats

Utilisez la commande `.import` pour importer des données CSV (valeurs séparées par des virgules) ou des données délimitées de manière similaire dans une table SQLite. La commande `.import` prend deux arguments qui sont la source à partir de laquelle les données doivent être lues et le nom de la table SQLite dans laquelle les données doivent être insérées. L'argument source est le nom d'un fichier à lire ou, s'il commence par un `|` caractère, il spécifie une commande qui sera exécutée pour produire les données d'entrée.

Notez qu'il peut être important de définir le `mode` avant d'exécuter la commande `.import`. Il est prudent d'empêcher le shell de ligne de commande d'essayer d'interpréter le texte du fichier d'entrée dans un format autre que la façon dont le fichier est structuré. Si les options `--csv` ou `--ascii` sont utilisées, elles contrôlent les délimiteurs d'entrée d'importation. Dans le cas contraire, les délimiteurs sont ceux en vigueur pour le mode de sortie en cours.

Pour importer dans une table ne figurant pas dans le schéma `principal`, l'option `--schema` peut être utilisée pour spécifier que la table se trouve dans un autre schéma. Cela peut être utile pour les bases de données attachées ou pour importer dans une table `TEMP`.

Lorsque `.import` est exécuté, le traitement de la première ligne d'entrée dépend de l'existence ou non de la table cible. S'il n'existe pas, le tableau est automatiquement créé et le contenu de la première ligne de saisie est utilisé pour définir le nom de toutes les colonnes du tableau. Dans ce cas, le contenu des données du tableau est extrait de la deuxième ligne d'entrée et des suivantes. Si la table cible existe déjà, chaque ligne de l'entrée, y compris la première, est considérée comme le contenu réel des données. Si le fichier d'entrée contient une ligne initiale d'étiquettes de colonnes, vous pouvez faire en sorte que la commande .import ignore cette ligne initiale en utilisant l'option `--skip 1`.

Voici un exemple d'utilisation, chargeant une table temporaire préexistante à partir d'un fichier CSV dont la première ligne contient des noms de colonnes :

```
sqlite> .import --csv --skip 1 --schema temp C:/work/data.csv tab1
```

Lors de la lecture des données d'entrée dans des modes autres que `ascii`, `.import` interprète l'entrée comme des enregistrements composés de champs conformément à la spécification *RFC 4180* avec cette exception : l'enregistrement d'entrée et les séparateurs de champ sont tels que définis par le mode ou par l'utilisation du Commande `.separator`. Les champs sont toujours sujets à la suppression des guillemets pour inverser les guillemets effectués conformément à la *RFC 4180*, sauf en mode ascii.

Pour importer des données avec des délimiteurs arbitraires et sans guillemets, définissez d'abord le mode ascii (`.mode ascii`), puis définissez les délimiteurs de champ et d'enregistrement à l'aide de la commande `.separator`. Cela supprimera la déquotation. Lors de `.import`, les données seront divisées en champs et enregistrements selon les délimiteurs ainsi spécifiés.

8.6 Exporter au format CSV

Pour exporter une table SQLite (ou une partie de table) au format CSV, définissez simplement le `mode` sur `csv`, puis exécutez une requête pour extraire les lignes souhaitées de la table. La sortie sera formatée au format CSV conformément à la *RFC 4180*.

```
sqlite> .headers sur
sqlite> .mode csv
sqlite> .once c:/work/dataout.csv
sqlite> SELECT * FROM tab1;
sqlite> .system c:/work/dataout.csv
```

Dans l'exemple ci-dessus, la ligne `.headers on` entraîne l'impression des étiquettes de colonnes comme première ligne de sortie. Cela signifie que la première ligne du fichier CSV résultant contiendra des étiquettes de colonnes. Si les étiquettes de colonnes ne sont pas souhaitées, désactivez plutôt `.headers off`. (Le paramètre `.headers off` est le paramètre par défaut et peut être omis si les en-têtes n'ont pas été préalablement activés.)

La ligne `.once FILENAME` fait que toutes les sorties de requête sont envoyées dans le fichier nommé au lieu d'être imprimées sur la console. Dans l'exemple ci-dessus, cette ligne entraîne l'écriture du contenu CSV dans un fichier nommé `C:/work/dataout.csv`.

La dernière ligne de l'exemple (le `.system c:/work/dataout.csv`) a le même effet qu'un double-clic sur le fichier `c:/work/dataout.csv` dans Windows. Cela fera généralement apparaître un tableur pour afficher le fichier CSV.

Cette commande ne fonctionne que telle qu'elle est écrite sous Windows. La ligne équivalente sur un Mac serait :

```
sqlite> .system oppen dataout.csv
```

Sous Linux et autres systèmes Unix, vous devrez saisir quelque chose comme :

```
sqlite> .system xdg-open dataout.csv
```

8.6.1 Exporter vers Excel

Pour simplifier l'exportation vers une feuille de calcul, l'interface en ligne de commande fournit la commande `.excel` qui capture le résultat d'une seule requête et envoie ce résultat au tableur par défaut sur l'ordinateur hôte. Utilisez-le comme ceci :

```
sqlite> .excel
sqlite> onglet SELECT * FROM ;
```

La commande ci-dessus écrit le résultat de la requête au format CSV dans un fichier temporaire, appelle le gestionnaire par défaut pour les fichiers CSV (généralement le tableur préféré tel qu'Excel ou LibreOffice), puis supprime le fichier temporaire. Il s'agit essentiellement d'une méthode abrégée permettant d'exécuter la séquence de commandes `.csv`, `.once` et `.system` décrite ci-dessus.

La commande `.excel` est en réalité un alias pour `.once -x`. L'option `-x` de .once lui permet d'écrire les résultats au format CSV dans un fichier temporaire nommé avec le suffixe `.csv`, puis d'appeler le gestionnaire par défaut du système pour les fichiers CSV.

Il existe également une commande `.once -e` qui fonctionne de manière similaire, sauf qu'elle nomme le fichier temporaire avec un suffixe `.txt` afin que l'éditeur de texte par défaut du système soit invoqué, au lieu de la feuille de calcul par défaut.

8.6.2 Exporter vers TSV (valeurs séparées par des tabulations)

L'exportation vers du TSV pur, sans aucune citation de champ, peut être effectuée en saisir `.mode tabs` avant d'exécuter une requête. Cependant, la sortie sera ne pas être lu correctement en mode onglets par la commande `.import` s'il contient des guillemets doubles. Pour obtenir le TSV cité selon la *RFC 4180* afin que il peut être saisi en mode onglets avec `.import`, entrez d'abord `.mode csv`, puis entrez `.separator \t` avant d'exécuter une requête.

9 Accès aux bases sous forme compressée

En plus de lire et d'écrire des fichiers de base de données SQLite, le Le programme `sqlite3` lira et écrira également les archives ZIP. Simplement spécifier un nom de fichier d'archive `ZIP` à la place d'un nom de fichier de base de données SQLite sur la ligne de commande initiale, ou dans la commande `.open`, et `sqlite3` détectera automatiquement que le fichier est une archive `ZIP` au lieu d'une base de données SQLite et l'ouvrira comme telle. Ceci fonctionne quel que soit le suffixe du fichier. Vous pouvez donc ouvrir des fichier `JAR`, `DOCX` et `ODP` ou tous les autres formats de fichier qui sont sont en réalité une archive `ZIP`.

Une archive `ZIP` semble être une base de données contenant une seule table avec le schéma suivant :

```
CREATE TABLE zip(
  name,     // Name of the file
  mode,     // Unix-style file permissions
  mtime,    // Timestamp, seconds since 1970
  sz,       // File size after decompression
  rawdata,  // Raw compressed file data
  data,     // Uncompressed file content
  method    // ZIP compression method code
);
```

Ainsi, par exemple, si vous vouliez voir l'efficacité de la compression (exprimée par la taille du contenu compressé par rapport au taille originale du fichier non compressé) pour tous les fichiers de l'archive ZIP, trié du plus compressé au moins compressé, vous pouvez exécuter une requête comme ça:

```
sqlite> SELECT name, (100.0*length(rawdata))/sz FROM zip ORDER BY 2;
```

Ou en utilisant les fonctions d'E/S de fichiers, vous pouvez extraire des éléments de l'archive ZIP :

```
sqlite> SELECT writefile(name,content) FROM zip
   ...> WHERE name LIKE 'docProps/%';
```

9.1 Details de l'implémentation de l'accès aux archives ZIP

Le shell de ligne de commande utilise la table virtuelle `Zipfile` pour accéder aux archives ZIP. Vous pouvez le voir en exécutant la commande `.schema` lorsqu'une archive `ZIP` est ouverte :

```
sqlite> .schema
CREATE VIRTUAL TABLE zip USING zipfile('document.docx')
/* zip(name,mode,mtime,sz,rawdata,data,method) */;
```

Lors de l'ouverture d'un fichier, si le client en ligne de commande découvre que le fichier est une archive `ZIP` au lieu d'une base de données SQLite, il ouvre en fait une base de données en mémoire, puis dans cette base de données en mémoire, il crée une instance de la table virtuelle `Zipfile` qui est attachée à l'archive ZIP.

Le traitement spécial pour ouvrir les archives `ZIP` est une astuce du shell de ligne de commande et non de la bibliothèque SQLite. Donc si vous voulez ouvrir une archive `ZIP` comme base de données dans votre application, vous devrez activer le module de table virtuelle `Zipfile` puis exécuter une instruction `CREATE VIRTUAL TABLE` appropriée.

10 Conversion d'une base en fichier texte

Utilisez la commande `.dump` pour convertir l'intégralité du contenu d'une base de données dans un seul fichier texte UTF-8. Ce fichier peut être reconverti en un base de données en la redirigeant vers `sqlite3`.

Une bonne façon de créer une copie d'archive d'une base de données est la suivante :

```
$ sqlite3 ex1 .dump | gzip -c >ex1.dump.gz
```

Cela génère un fichier nommé `ex1.dump.gz` qui contient tout ce dont vous avez besoin pour reconstruire la base de données ultérieurement ou sur une autre machine. Pour reconstruire la base de données, tapez simplement :

```
$ zcat ex1.dump.gz | sqlite3 ex2
```

Le format de texte est du SQL pur, vous pouvez donc également utiliser la commande `.dump` pour exporter une base de données SQLite vers d'autres moteurs de base de données SQL populaires. Comme ceci:

```
$ createdb ex2
$ sqlite3 ex1 .dump | psql ex2
```

11 Récupérer une base de données corrompue

Comme la commande `.dump`, la commande `.recover` tente de convertir l'intégralité du contenu d'un fichier de la base de données en texte. La différence est qu'au lieu de lire les données à l'aide de l'interface de base de données SQL normale, `.recover` tente de réassembler la base de données en fonction des données extraites directement du plus grand nombre possible de pages de base de données. Si la base de données est corrompue, `.recover` est généralement capable de récupérer les données de toutes les parties non corrompues de la base de données, alors que `.dump` s'arrête dès les premiers signes de corruption.

Si la commande `.recover` récupère une ou plusieurs lignes qu'elle ne peut pas attribut à n'importe quelle table de base de données, le script de sortie crée un Table `lost_and_found` pour stocker les lignes orphelines. Le schéma de la table loss_and_found est la suivante :

```
CREATE TABLE lost_and_found(
    rootpgno INTEGER,   -- page racine de l'arbre pgno,
    pgno INTEGER,       -- numéro de page contenant la rangée,
    nfield INTEGER,     -- numéro du champs dans la rangée,
    id INTEGER,         -- valeur du champs rowid, ou NULL
    c0, c1, c2, ...     -- les colonnes des champs de la rangée
);
```

La table `lost_and_found` contient une ligne pour chaque ligne orpheline récupérée de la base de données. De plus, il y a une ligne pour chaque entrée d'index récupérée qui ne peut être attribuée à aucun index SQL. En effet, dans une base de données SQLite, le même format est utilisé pour stocker les entrées d'index SQL et SANS les entrées de table ROWID.

Column	Contenu
rootpgno	Même s'il n'est pas possible d'attribuer la ligne à une table de base de données spécifique, elle peut faire partie d'une structure arborescente au sein de la fichier de base de données. Dans ce cas, le numéro de page racine de cette arborescence est stocké dans cette colonne. Ou, si la page sur laquelle la ligne a été trouvée n'est pas faisant partie d'une arborescence, cette colonne stocke une copie de la valeur dans column `pgno` - le numéro de page de la page sur laquelle la ligne a été trouvée. Dans de nombreux cas, mais pas tous, toutes les lignes de la table loss_and_found avec la même valeur dans cette colonne appartient à la même table.
pgno	Le numéro de la page sur laquelle on trouve cette rangée..
nfield	Le nombre de champs dans cette rangée.
id	Si la ligne provient d'une table SANS ROWID, cette colonne contient NUL. Sinon, il contient la valeur entière rowid de 64 bits pour la ligne.

c0, c1, c2... Les valeurs de chaque colonne de la ligne sont stockées dans ces colonnes. La commande `.recover` crée la table loss_and_found avec autant de colonnes comme requis par la ligne orpheline la plus longue.

Si le schéma de base de données récupéré contient déjà une table nommée `lost_and_found`, la commande `.recover` utilise le nom `lost_and_found0`. Si le nom `lost_and_found0` est également déjà pris, `lost_and_found1`, et ainsi de suite. Le nom par défaut `lost_and_found` peut être remplacé en appelant `.recover` avec le commutateur `--lost-and-found`. Pour exemple, pour que le script de sortie appelle la table `orphaned_rows` :

```
sqlite> .recover --lost-and-found orphaned_rows
```

12 Charger des Extensions

Vous pouvez ajouter de nouvelles fonctions SQL personnalisées, des séquences de classement, des tables virtuelles et VFS vers le shell de ligne de commande au moment de l'exécution à l'aide de la commande `.load`. D'abord, construire l'extension en tant que DLL ou bibliothèque partagée (comme décrit dans le Document Extensions chargeables au moment de l'exécution) puis tapez :

```
sqlite> .load /path/to/my_extension
```

Notez que SQLite ajoute automatiquement le suffixe d'extension approprié (`.dll` sur Windows, `.dylib` sur Mac, `.so` sur la plupart des autres Unix) au nom de fichier d'extension. C'est généralement une bonne idée de préciser l'intégralité chemin d'accès à l'extension.

SQLite calcule le point d'entrée de l'extension en fonction de l'extension nom de fichier. Pour remplacer ce choix, ajoutez simplement le nom de l'extension comme deuxième argument de la commande `.load`.

Le code source de plusieurs extensions utiles peut être trouvé dans le Sous-répertoire `ext/misc` de l'arborescence des sources SQLite. Vous pouvez utiliser ces extensions telles quelles, ou comme base pour créer votre propre personnalisation extensions pour répondre à vos propres besoins particuliers.

13 Hachages cryptographiques

La commande point `.sha3sum` calcule un hâchage *SHA3* du *contenu* de la base de données. Pour être clair, le hâchage est calculé sur le contenu de la base de données, pas sur sa représentation sur le disque. Par exemple, ceci signifie qu'un `VACUUM` ou une transformation similaire de préservation des données ne change pas le hâchage.

La commande `.sha3sum` prend en charge les options `--sha3-224`, `--sha3-256`, `--sha3-384` et `--sha3-512` pour définir la variété de *SHA3* à utiliser pour le hâchage. La valeur par défaut est *SHA3-256*.

Le schéma de la base de données (dans la table `sqlite_schema`) n'est normalement pas inclus dans le hachage, mais peut être ajouté par l'option `--schema`.

La commande `.sha3sum` prend un seul argument facultatif similaire au motif `LIKE`. Si cette option est présente, seules les tables dont les noms correspondent au motif seront hachées.

La commande `.sha3sum` est implémentée à l'aide du fonction d'extension `sha3_query()` incluse avec le shell de ligne de commande.

14 Autotests du contenu de la base de données

La commande `.selftest` tente de vérifier si une base de données est intacte et non corrompue. La commande `.selftest` recherche une table dans le schéma nommé `selftest` et définie comme suit :

```
CREATE TABLE selftest(
  tno INTEGER PRIMARY KEY,  -- Test number
  op TEXT,                  -- 'run' or 'memo'
  cmd TEXT,                 -- SQL command to run, or text of "memo"
  ans TEXT                  -- Expected result of the SQL command
);
```

La commande `.selftest` lit les lignes de la table `selftest` en suivant l'ordre `selftest.tno`. Pour chaque ligne `memo`, il écrit le texte dans `cmd` dans le résultat. Pour chaque ligne inspectée, il exécute l'instruction SQL `cmd` et compare le résultat à la valeur dans `ans` et affiche un message d'erreur si les résultats diffèrent.

S'il n'y a pas de table d'autotest, la commande `.selftest` exécute un pragma de vérification d'intégrité.

La commande `.selftest --init` crée la table d'autotest si elle n'existe pas déjà le cas. Puis elle ajoute des entrées qui vérifient le hachage SHA3 du contenu de tous les tableaux. Les exécutions ultérieures de `.selftest` vérifieront que la base de données n'a pas été modifiée. Pour générer des tests pour vérifiez qu'un sous-ensemble des tables est inchangé, exécutez simplement `.selftest --init` puis supprimez les lignes d'autotest qui font référence aux tables et qui ne sont pas constantes.

15 Prise en charge des archives SQLite

La commande `.archive` et l'option de ligne de commande `-A"` fournissent une prise en charge intégrée du format d'archive de SQLite. L'interface est similaire à celui de la commande `tar` sur les systèmes Unix. Chaque invocation de la commande `.ar` ne doit spécifier qu'une seule option. Voici les commandes disponibles pour `.archive` :

Option	Option longue	usage
`-c`	`--create`	Créé une nouvelle archive contenant le fichier spécifié.
`-x`	`--extract`	Extrait le fichier spécifié de l'archive.
`-i`	`--insert`	Ajoute des fichiers à une archive existante
`-r`	`--remove`	Enlève des fichiers d'une archive.
`-t`	`--list`	Liste les fichers de l'archive.
`-u`	`--update`	Ajoute des fichiers à une archive existante *s'ils* ont été changés.

En plus de l'option de commande, chaque invocation de `.ar` peut spécifier une ou plusieurs options de modification. Certaines options de modification nécessitent un argument, d'autres non. Les options de modification suivantes sont disponibles :

Option	Option_longue	Usage
`-v`	`--verbose`	Liste chaque fichier en cours de traitement.
`-f FILE`	`--file FILE`	Si spécifié, utilisez le fichier `FILE` comme archive. Sinon, supposons que la base de données `"principale"` (ndt. “main”) actuelle est l'archive à exploiter.
`-a FILE`	`--append FILE`	Comme l'option `--file`, utilise le fichier `FILE` comme archive, mais ouvre le fichier en utilisant la VFS `apndvfs` pour que l'archivage soit ajouté à la fin du fichier `FILE` s'il existe.
`-C DIR`	`--directory DIR`	Si spécifié, interprète tous les chemins relatifs par rapport à `DIR`, au lieu d'utiliser le répertoire courant.
`-g`	`--glob`	Utilise `glob(Y,X)` pour faire correspondre les arguments à des noms dans l'archive.
`-n`	`--dryrun`	Montre l'instruction SQL, qui devrait être exécutée pour réaliser l'opération d'archivage, mais dans les faits ne change rien.
`--`	`--`	Tous les mots qui suivent sur la ligne de commande sont des arguments de commande, et plus des options.

Pour une utilisation en ligne de commande, ajoutez les options de ligne de commande de style court immédiatement après le `-A`, sans espace intermédiaire. Tous les arguments suivants sont considérés comme faisant partie de la commande `.archive`. Par exemple, les commandes suivantes sont équivalentes :

```
sqlite3 new_archive.db -Acv file1 file2 file3
sqlite3 new_archive.db ".ar -cv file1 file2 file3"
```

Les options de style long et court peuvent être mélangées. Par exemple, ces instrucitons sont équivalentes :

```
-- Deux méthodes pour créer un nouvelle archive "new_archive.db"
-- contenant les fichiers "file1", "file2" et "file3".
.ar -c --file new_archive.db file1 file2 file3
.ar -f new_archive.db --create file1 file2 file3
```

Alternativement, le premier argument suivant `.ar` peut être la concaténation de la forme courte de toutes les options requises (sans le caractère `-`). Dans ce cas, les arguments des options qui les nécessitent sont lus dans les mots qui suivent sur la ligne de commande, et tous les mots restants sont compris comme les arguments de commande. Par exemple:

```
-- Créé une nouvelle archive "new_archive.db" contenant les fichiers
-- "file1" et "file2" dans le répertoire "dir1".
.ar cCf dir1 new_archive.db file1 file2 file3
```

15.1 La commande de création d'archive (.ar –create)

La commande `.ar --create` crée une nouvelle archive en écrasant toute archive existante (soit dans le base de données "principale" actuelle ou dans le fichier spécifié par une option `--file`). Chaque argument qui suit les options est un fichier à ajouter à l'archive. Les répertoires sont importés de manière récursive. Voir ci-dessus pour des exemples.

15.2 La commande d'extraction (.ar –extract)

La commande `.ar --extract` extrait les fichiers de l'archive (soit vers le répertoire de travail actuel, soit vers le répertoire spécifié par une option `--directory`). Les fichiers ou les répertoires dont les noms correspondent aux arguments affectés par l'option `--glob` sont extraits. Si aucun argument ne suit les options, tous les fichiers et les répertoires sont extraits. Tous les répertoires spécifiés sont extraits récursivement. C'est une erreur si des noms spécifiés ou des modèles de recherche ne correspondent à rien dans l'archive.

```
-- Extrait tous les fichiers de l'archive de la base courante
-- "main" dans le répertoire courant. Liste les fichiers pendant
-- leur extraction.
.ar --extract --verbose

-- Extrait le fichier "file1" de l'archive "ar.db" dans le
-- répertoire "dir1".
.ar fCx ar.db dir1 file1

-- Extrait les fichiers avec l'extension ".h" dans le répertoire
-- "headers".
.ar -gCx headers *.h
```

15.3 La commande pour lister le contenu d'une archive (.ar –list)

La commande `.ar --list` répertorie le contenu de l'archive. Si aucun argument n'est spécifié, alors tous les fichiers sont répertoriés. Sinon, seuls ceux qui correspondent aux arguments de l'option `--glob` sont listés. Actuellement, l'option `--verbose` ne modifie pas le comportement de cette commande. Cela pourrait changer dans l'avenir.

```
-- Liste le contenu de l'archive dans la base courante "main"..
.ar --list
```

15.4 Les commandes pour insérer et mettre à jour (.ar –insert/.ar –update)

Les commandes `.ar --update` et `.ar --insert` fonctionnent comme la commande `.ar --create`, sauf qu'ils ne suppriment pas l'archive courante avant de commencer. Les nouvelles versions des fichiers remplacent silencieusement les fichiers existants portant les mêmes noms, mais sinon le contenu initial des archives (le cas échéant) restent intactes.

Pour la commande `.ar --insert`, tous les fichiers répertoriés sont insérés dans le archive. Pour la commande `.ar --update`, les fichiers ne sont insérés que s'ils ne pré-existent pas dans l'archive, ou que si leur `mtime` ou `mode` est différent de ce qui se trouve actuellement dans l'archive.

Remarque de compatibilité : avant SQLiteversion 3.28.0 (2019-04-16), seul l'option `--update` était prise en charge mais cette option fonctionnait comme `--insert`, car il réinsérait le fichier, qu'il ait été modifié ou non.

15.5 La commande pour enlever (.ar –remove)

La commande `.ar --remove` supprime les fichiers et les répertoires qui correspondent aux arguments fournis (le cas échéant) par l'option `--glob`. C'est un erreur de fournir des arguments qui ne correspondent à rien dans l'archive.

15.6 Les operations sur les archives ZIP

Si `FILE` est une archive `ZIP` plutôt qu'une archive SQLite, la commande `.archive` et l'option de ligne de commande `-A` fonctionnent toujours. C'est réalisé en utilisant l'extension zipfile (`https://www.sqlite.org/zipfile.html`). Les commandes qui suivent sont à peu près équivalentes et ne diffèrent que par le format de la sortie :

Traditional Command	Commande équivalente avec sqlite3.exe
`unzip archive.zip`	`sqlite3 -Axf archive.zip`
`unzip -l archive.zip`	`sqlite3 -Atvf archive.zip`
`zip -r archive2.zip dir`	`sqlite3 -Acf archive2.zip dir`

15.7 Les Instructions SQL réalisant des opérations d'archivage SQLite

Les différentes commandes d'archivage de SQLite sont implémentées à l'aide d'instruction SQL. Les développeurs d'applications peuvent ajouter simplement un support pour la lecture et l'écriture des archives SQLite dans leurs projets en utilisant des instructions SQL approprié.

Pour voir quelles instructions SQL sont utilisées pour implémenter une archive SQLite opération, ajoutez l'option `--dryrun` ou `-n`. Ceci affiche les instructions SQL affichées mais inhibe leur exécution.

Les instructions SQL implémentant les opérations d'archivage SQLite utilisent diverses extensions chargeables. Ces extensions sont toutes disponibles dans l'arborescence des sources de SQLite dans les sous répertoires de `ext/misc/`. Les extensions nécessaires à la prise en charge complète de l'archivage SQLite incluent:

1. fileio.c — Cette extension ajoute les fonctions SQL `readfile()` et `writefile()` pour lire et écrire à partir de fichiers présents sur le disque. L'extension `fileio.c` inclut également la fonction `fsdir()` pour répertorier le contenu d'un répertoire et la fonction `lsmode()` qui convertit les codes numérique `st_mode` de l'appel système `stat()` en chaînes lisibles par l'homme après, à la manière de la commande `ls -l`.
2. sqlar.c — Cette extension ajoute les fonctions `sqlar_compress()` et `sqlar_uncompress()` nécessaires pour compresser et décompresser un fichier au fur et à mesure de son extraction d'une archive SQLite.
3. zipfile.c — Cette extension implémente la fonction `zipfile(FILE)` qui est utilisée pour lire les archives `ZIP`. Cette extension n'est nécessaire que pour lire les archives `ZIP` au lieu des archives SQLite.
4. appendvfs.c — Cette extension implémente un système de fichiers virtuel (VFS) supplémentaire qui permet d'ajouter une base de données SQLite à un autre fichier, comme un exécutable. Cette extension n'est nécessaire que si l'option `--append` de la commande `.archive` est utilisée.

16 Paramètres SQL

SQLite permet aux paramètres liés d'apparaître dans une instruction SQL partout où une valeur littérale est permise. Les valeurs de ces les paramètres sont définis à l'aide de la famille d'API sqlite3_bind_...().

Les paramètres peuvent être nommés ou non. Un paramètre sans nom est un un point d'interrogation seul (?). Les paramètres nommés sont un ? suivis immédiatement par un chiffre (ex : ?15 ou ?123) ou un des caractères $, :, ou @ suivi d'un nom alphanumérique (ex : $var1, :xyz, @bingo).

Ce shell en ligne de commande laisse les paramètres sans nom détachés, ce qui signifie qu'ils auront une valeur SQL NULL. Par contre, les paramètres nommés auront des valeurs assignées. S'il existe une table TEMP nommée sqlite_parameters avec un schéma comme celui-ci :

```
CREATE TEMP TABLE sqlite_parameters(
  key TEXT PRIMARY KEY,
  value
) WITHOUT ROWID;
```

Et s'il y a une entrée dans cette table où la colonne des identifiants correspond exactement au nom du paramètre (comprenant les initiales ?, $, : ou le caractère @), le paramètre reçoit alors la valeur de la colonne des valeurs correspondantes. Si aucune entrée n'existe, le paramètre par défaut est NULL.

La commande .parameter existe pour simplifier la gestion de cette table. La commande .parameter init (souvent abrégée en .param init) crée la table temp.sqlite_parameters si elle n'existe pas déjà. La commande .param list affiche toutes les entrées de la table temp.sqlite_parameters. La commande .param clear supprime le table temp.sqlite_parameters. Les commandes .param set CLÉ VALEUR et .param unset CLÉ créent ou suppriment des entrées du table temp.sqlite_parameters.

La VALEUR transmise à .param set CLÉ VALEUR peut être soit un littéral SQL ou toute autre expression ou requête SQL qui peut être évaluée pour générer une valeur. Ceci permet de définir des valeurs de différents types. Si ce genre d'évaluation échoue, la VALEUR fournie est mise sous guillemets et insérée comme un texte. Parce qu'une telle évaluation peut échouer en fonction du contenu de la VALEUR, le moyen fiable d'obtenir une valeur de texte est de l'entourer avec des guillemets simples qui la protègeront pendant l'analyse décrite ci-dessous. Par exemple, (sauf si l'on souhaite une valeur de -1365) :

```
.parameter init
.parameter set @phoneNumber "'202-456-1111'"
```

Notez que les guillemets doubles servent à protéger les guillemets simples et de s'assurer ainsi que le texte cité est analysé comme un seul argument.

La table temp.sqlite_parameters fournit uniquement des valeurs pour les paramètres dans le shell de la ligne de commande. La table temp.sqlite_parameter n'a aucun effet sur requêtes exécutées directement à l'aide de l'API SQLite du langage C. Les applications individuelles sont censées implémenter leur propres paramètres obligatoires. Vous pouvez rechercher sqlite_parameters dans le code source de l'interface en ligne de commande de SQLite pour voir comment l'interface en ligne de commande établit la liaison avec les paramètres pour l'utiliser comme guide pour les implémenter dans vos propres applications.

17 Recommendation pour les experts SQLite sur les index

Note: This command is experimental. It may be removed or the interface modified in incompatible ways at some point in the future.

Pour la plutpart des bases de données non triviales, la clé pour une bonne performance est de créer les bons index. Dans ce contexte, les « bons index SQL » signifient ceux qui feront que les requêtes demandées par une application seront rapides. La commande `.expert` peut aider à définir les meilleurs index pour une requête spécifique, si elles sont présentes dans la base de données.

La commande `.expert` est utilisée dans un premier temps, suivie de la requète SQL sur une ligne séparée. Par exemple, considérez la session suivante :

```
sqlite> CREATE TABLE x1(a, b, c);
sqlite> .expert
sqlite> SELECT * FROM x1 WHERE a=? AND b>?;
CREATE INDEX x1_idx_000123a7 ON x1(a, b);

0|0|0|SEARCH TABLE x1 USING INDEX x1_idx_000123a7 (a=? AND b>?)

sqlite> CREATE INDEX x1ab ON x1(a, b);
sqlite> .expert
sqlite> SELECT * FROM x1 WHERE a=? AND b>?;
(no new indexes)

0|0|0|SEARCH TABLE x1 USING INDEX x1ab (a=? AND b>?)
```

Dans l'exemple ci-dessus, l'utilisateur crée un schéma de base de données (une table simple `x1`) et ensuite utilise la commande `.expert` pour analyser une requête, dans cas `"SELECT * FROM x1 WHERE a=? AND b>?"`. L'outil shell recommande alors que l'utilisateur crée un nouvel index (index `"x1_idx_000123a7`) et sort le plan que la requète utiliserait avec le format `EXPLAIN QUERY PLAN`. L'utilisateur crée alors un index équivalent avec un schéma équivalent et lance l'analyse de nouveau sur la même requète. Cette fois, l'outil shell ne recommande par un nouvel index et renvoie les plan que SQLite va utiliser pour cette requète en utilisant l'index donné.

La commande `.expert` accepte les options suivantes :

Option	Objet
`--verbose`	Si présent, la sortie donnera un rapport plus verveux pour chaque requète analysée.

`--sample POURCENT` Ce paramètre à 0 comme valeur par défaut, ce qui fait que la command `.expert` ne recommande que des index basés sur des requètes et le schéma de base données. Ce paramètre est par défaut à 0, ce qui fait que la commande `.expert` recommande des index basés uniquement sur la requête et le schéma de la base de données. Ceci est similaire à la façon dont le planificateur de requêtes SQLite sélectionne les index pour les requêtes si l'utilisateur n'a pas exécuté la commande `ANALYZE` sur la base de données pour générer des statistiques de distribution des données. Si cette option reçoit un argument non nul, la commande `.expert` génère des statistiques de distribution de données similaires pour tous les index considérés, basées sur un pourcentage (`PERCENT`) des lignes actuellement stockées dans chaque table de la base de données. Pour les bases de données avec des distributions de données inhabituelles, cela peut conduire à de meilleures recommandations d'index, en particulier si l'application prévoit d'exécuter `ANALYZE`. Pour les petites bases de données et les processeurs modernes, il n'y a généralement aucune raison de ne pas passer `"--sample 100"`. Cependant, la collecte de statistiques de distribution des données peut être coûteuse pour les grandes tables de base de données. Si l'opération est trop lente, essayez de passer une valeur plus petite pour l'option `--sample`.

La fonctionnalité décrite dans cette section peut être intégrée dans d'autres applications ou outils en utilisant le code d'extension SQLite expert.

Un schéma de base de données qui incorpore des fonctions SQL personnalisées disponibles via le mécanisme de chargement d'extension peut nécessiter des dispositions spéciales pour fonctionner avec la fonctionnalité `.expert`. Étant donné que cette fonctionnalité utilise des connexions supplémentaires pour implémenter ses fonctionnalités, ces fonctions personnalisées doivent être rendues disponibles pour ces connexions supplémentaires. Cela peut être fait grâce aux options de chargement/utilisation d'extension décrites dans les sections Chargement Automatique des Extensions Liées Statiquement et Extensions Chargeables Persistantes.

18 Travailler avec des connexions multiples à base de données

À partir de la version 3.37.0 (27-11-2021), l'interface en ligne de commande a la capacité de maintenir plusieurs connexions de bases de données ouvertes en même temps. Une seule connexion de base de données est active à la fois. Les connexions inactives restent ouvertes mais sont en attente.

Utilisez la commande point `.connection` (souvent abrégée en `.conn`) pour voir une liste des connexions de bases de données et une indication de laquelle est actuellement active. Chaque connexion de base de données est identifiée par un entier entre 0 et 9. (Il peut y avoir au maximum 10 connexions ouvertes simultanément.) Passez à une autre connexion de base de données, en la créant si elle n'existe pas déjà, en tapant la commande `.conn` suivie de son numéro. Fermez une connexion de base de données en tapant `.conn close N` où `N` est le numéro de la connexion.

Bien que les connexions de bases de données SQLite sous-jacentes soient complètement indépendantes les unes des autres, de nombreux paramètres de l'interface en ligne de commande, tels que le format de sortie, sont partagés entre toutes les connexions de bases de données. Ainsi, changer le mode de sortie dans une connexion le changera dans toutes. En revanche, certaines commandes point, comme `.open`, n'affectent que la connexion actuelle.

19 Fonctionalités diverses de l'extension

L'interface en ligne de commande est construit avec plusieurs extensions `SQLite` qui ne sont pas incluses avec la bibliothèque `SQLite`. Quelques-unes ajoutent des fonctionnalités non décrites dans les sections précédentes, à savoir :

- la séquence de collation `UINT` qui traite les entiers non signés intégrés dans du texte selon leur valeur, ainsi que d'autres textes, pour le tri ;
- l'arithmétique décimale fournie par l'extension `decimal` ;
- les `generate_series()` de la fonction `table-valued`;
- les fonctions `base64()` et `base85()` qui encodent un blob en texte *base64* ou *base85*, ou décodent le même en blob ;
- le support des expressions régulières étendues POSIX liées à l'opérateur `REGEXP`.

20 Autres commandes point

Il y a beaucoup d'autres commandes point disponibles dans le shell en ligne de commande. Voir la commandes `.help` pour une liste complète pour toutes les versions de SQLite.

21 Utiliser sqlite3 dans un script shell

Une façon d'utiliser `sqlite3` dans un script shell consiste à utiliser `echo` ou `cat` pour générer une séquence de commandes dans un fichier, puis appeler `sqlite3` pendant rediriger l'entrée du fichier de commande généré. Cela fonctionne bien et est approprié dans de nombreuses circonstances. Mais pour plus de commodité, sqlite3 permet de saisir une seule commande SQL sur la ligne de commande comme un deuxième argument après le nom de la base de données. Lorsque le programme `sqlite3` est lancé avec deux arguments, le deuxième argument est passé au SQLitebibliothèque pour le traitement, les résultats de la requête sont imprimés sur la sortie standard en mode liste et le programme se termine. Ce mécanisme est conçu pour faire sqlite3 facile à utiliser avec des programmes comme `awk`. Pour exemple:

```
$ sqlite3 ex1 'select * from tbl1' \
>  | awk '{printf "<tr><td>%s<td>%s\n",$1,$2 }'
<tr><td>hello<td>10
<tr><td>goodbye<td>20
$
```

22 Marquer la fin d'une instruction SQL

Les commandes SQLite sont normalement terminée par un point virgule. Dans l'interface en ligne de commande, vous pouvez aussi utiliser le mot `GO` (insensible à la casse) ou la barre oblique `/` sur une ligne pour terminer une commande. Ces caractères sont utilisés par, respectivement, SQL Server et Oracle et ils sont supporté par l'interface en ligne de commande de SQLite pour des raisons de compatibilités. Ceci ne fonctionne pas avec sqlite3_exec(), car l'interface en ligne de commande traduit ces entrées en point virgules avant des les envoyer au cœur d'exécution de `SQLite`.

23 Options de la ligne de commandes

Il y a beaucoup d'optiopns pour les commandes de l'interface shell. Utilisez la commande `--help` pour en avoir la liste :

```
$ sqlite3 --help
Usage: ./sqlite3 [OPTIONS] FILENAME [SQL]
FILENAME is the name of an SQLite database. A new database is created
if the file does not previously exist. Defaults to :memory:.
OPTIONS include:
   --                   treat no subsequent arguments as options
   -A ARGS...           run ".archive ARGS" and exit
   -append              append the database to the end of the file
   -ascii               set output mode to 'ascii'
   -bail                stop after hitting an error
   -batch               force batch I/O
   -box                 set output mode to 'box'
   -column              set output mode to 'column'
   -cmd COMMAND         run "COMMAND" before reading stdin
   -csv                 set output mode to 'csv'
   -deserialize         open the database using sqlite3_deserialize()
   -echo                print inputs before execution
   -init FILENAME       read/process named file
   -[no]header          turn headers on or off
   -help                show this message
   -html                set output mode to HTML
   -interactive         force interactive I/O
   -json                set output mode to 'json'
   -line                set output mode to 'line'
   -list                set output mode to 'list'
   -lookaside SIZE N    use N entries of SZ bytes for lookaside memory
   -markdown            set output mode to 'markdown'
   -maxsize N           maximum size for a --deserialize database
   -memtrace            trace all memory allocations and deallocations
   -mmap N              default mmap size set to N
   -newline SEP         set output row separator. Default: '\n'
   -nofollow            refuse to open symbolic links to database files
   -nonce STRING        set the safe-mode escape nonce
   -nullvalue TEXT      set text string for NULL values. Default ''
   -pagecache SIZE N    use N slots of SZ bytes each for page cache mem
   -pcachetrace         trace all page cache operations
   -quote               set output mode to 'quote'
   -readonly            open the database read-only
   -safe                enable safe-mode
   -separator SEP       set output column separator. Default: '|'
   -stats               print memory stats before each finalize
   -table               set output mode to 'table'
   -tabs                set output mode to 'tabs'
```

```
-unsafe-testing      allow unsafe commands and modes for testing
-version             show SQLite version
-vfs NAME            use NAME as the default VFS
-zip                 open the file as a ZIP Archive
```

L'interface en ligne de commande est flexible pour ce qui concerne le formattage des options. On peut mettre un ou deux caractères `-` en préfixe. Donc `-box` et `--box` signifie la même chose. Les options en ligne de commande sont traitées de gauche à droite. Donc une option `--box` remplacera une option `--quote` qui la précèderait.

La plupart des options en lignes de commandes s'expliquent par elles mêmes, mais certaines méritent quelques explications supplémentaires ci-après.

23.1 L'option de ligne de commande `--safe`

L'option de ligne de commande `safe` tente de désactiver toutes les fonctionnalités du CLI pouvant entraîner des modifications sur l'ordinateur hôte autres que des modifications au fichier de base de données spécifique nommé sur la ligne de commande. L'idée est que si vous recevez un gros script SQL d'un inconnu ou non fiable source, vous pouvez exécuter ce script pour voir ce qu'il fait sans risquer de exploiter en utilisant l'option `safe`. L'option `safe` désactive (parmi autres choses):

- La commande `.open`, sauf is l'option `--hexdb` est utilisée ou si le nom du fichier est `:memory:`. This prevents the script from reading or writing any database files not named on the original command-line. Cela empêche le script de lire ou d'écrire des fichiers de base de données non nommés sur la ligne de commande d'origine.
- La commande SQL `ATTACH`.
- Les fonctions SQL qui peuvent potentiellement avoir des effets de bord dangereux comme `edit()`, `fts3_tokenizer()`, `load_extension()`, `readfile()` and `writefile()`.
- La commande `.archive`.
- Les commandes `.backup` et `.save`.
- La commandes `.import`.
- La commande `.load`.
- La commande `.log`.
- Les commandes `.shell` et `.system`.
- Les commandes `.excel`, `.once` et `.output`.
- Les autres commandes qui peuvent avoir des effets de bords délétères.

En pratique, toutes les fonctionnalités de l'interface en ligne de commande qui lisent ou écrivent à partir d'un fichier sur un disque autre que le fichier de base de données principal sont désactivé.

23.1.1 Court-circuiter des commandes particulières avec –safe restrictions

Si l'option `--nonce NONCE` est également incluse sur la ligne de commande, par exemple une chaîne `NONCE` volumineuse et arbitraire, puis la commande `.nonce NONCE` (avec la même

grande chaîne occasionnelle) autorisera la prochaine instruction SQL ou commande point pour contourner les restrictions `--safe`.

Supposons que vous souhaitiez exécuter un script suspect et que celui-ci nécessite une ou deux des fonctionnalités que `--safe désactive` normalement. Par exemple, supposons qu'il doive ATTACHER une base de données supplémentaire. Ou supposons que Le script doive charger une extension spécifique. Ceci peut être accompli en précédant l'instruction ATTACH (soigneusement vérifiée) ou la commande `.load` avec une commande `.nonce` appropriée et en fournissant la même valeur occasionnelle à l'aide de l'option en ligne de commande `--nonce`. Ces commandes spécifiques seront alors autorisées à s'exécuter normalement, mais toutes les autres commandes dangereuses seront toujours restreintes.

L'utilisation de `.nonce` est dangereuse car une erreur peut permettre à un script hostile d'endommager votre système. Par conséquent, n'utilisez `.nonce` qu'avec précaution, parcimonie, et en dernier recours lorsqu'il n'existe pas d'autres moyens d'obtenir un script à exécuter en mode `--safe`.

23.2 L'option de ligne de commande –unsafe-testing

L'option de ligne de commande `--unsafe-testing` permet d'utiliser l'interface en ligne de commande pour des tests internes de la bibliothèque SQLite. Ce n'est ni nécessaire ni utile pour l'utiliser en utilitaire pour créer, modifier ou interroger des bases de données SQLite. Il est prévu pour permettre des tests de scripts avec des changements de schéma, des défaites des mesures défensives et certains commandes point à usage spécial, non documentées.

Une mauvaise utilisation qui nécessiterait l'utilisation de l'option `--unsafe-testing` ne sera généralement pas considéré comme un boggue pour cette seule raison. Le comportement de la lgine de commande avec `--unsafe-testing` n'est ni pris en charge ni défini.

23.3 Les options de ligne de commande –no-utf8 et –utf8

Sur la plateforme Windows, lorsque la console est utilisée en entrée ou en sortie, une traduction est requise entre l'encodage de caractères disponible dans la console et à la représentation textuelle *UTF-8* interne à l'interface en ligne de commande. Les versions précédentes acceptaient ces options pour activer ou désactiver l'utilisation d'une traduction qui reposait sur une fonctionnalité de la console Windows grâce à laquelle elle pourrait être amené à produire ou à accepter *UTF-8* sur les versions modernes du système d'exploitation.

Les versions actuelles de l'interface (3.44.1 ou ultérieures) effectuent les E/S de la console en lisant ou en écrivant UTF-16 depuis/vers les API de la console Windows. Parce que cela fonctionne correctement même sur les versions de Windows remontant à Windows 2000, il y a plus besoin de ces options. Ils sont toujours acceptés, mais sans effet.

Dans tous les cas, les E/S texte hors console sont codées en UTF-8.

Sur les plateformes non Windows, ces options sont également ignorées.

24 Compiler le programme sqlite3 depuis ses sources

La commande de compilation en ligne de commande sur un shell sur les système Unix ou Windows avec MinGW fonctionne avec les commandes habituelles `configure/make` :

```
sh configure; make
```

Cette commande `configure/make` fonctionne que vous bâtissiez le programme de l'arbre des sources canonique oubien d'un paquet fusionné. Il y a quelque dépendances. Quand on compile à partir des sources canonique, une installation opérationnelle de `tclsh` est requise. Si vous utilisez un paquet fusionné, toutes les tâches amont faites normalement avec `tclsh` auront déjà été faite et seulement les outils normaux sont requis.

Un bibliothèque de compression `zlib` est nécessaire pour que la commande `.archive` puisse opérer.

Sous Windows avec `MSVC`, utilisez `nmake` avec le fichier `Makefile.msc` :

```
nmake /f Makefile.msc
```

Pour un fonctionnement correct avec la commande `.archive`, faites une copie du code source `zlib` dans le sous-répertoire `compat/zlib` de l'arbre des sources et compiler de la manière suivante :

```
nmake /f Makefile.msc USE_ZLIB=1
```

24.1 Tout compiler par soi-même

Le code source de l'interface de ligne de commande `sqlite3` se trouve dans un seul fichier nommé `shell.c`. Le fichier source `shell.c` est généré à partir d'autres sources, mais la plupart du code de `shell.c` se trouve dans `src/shell.c.in`. (Régénérez `shell.c` en tapant `make shell.c` à partir de l'arborescence des sources canoniques.) Compilez le fichier `shell.c` (avec le code source de la bibliothèque `sqlite3`) pour générer l'exécutable. Par exemple:

```
gcc -o sqlite3 shell.c sqlite3.c -ldl -lpthread -lz -lm
```

Les options de compilation supplémentaires suivantes sont recommandées afin de fournir un shell de ligne de commande complet :

- -DSQLITE_THREADSAFE=0
- -DSQLITE_ENABLE_EXPLAIN_COMMENTS
- -DSQLITE_HAVE_ZLIB
- -DSQLITE_INTROSPECTION_PRAGMAS
- -DSQLITE_ENABLE_UNKNOWN_SQL_FUNCTION
- -DSQLITE_ENABLE_STMTVTAB
- -DSQLITE_ENABLE_DBPAGE_VTAB
- -DSQLITE_ENABLE_DBSTAT_VTAB
- -DSQLITE_ENABLE_OFFSET_SQL_FUNC
- -DSQLITE_ENABLE_JSON1
- -DSQLITE_ENABLE_RTREE
- -DSQLITE_ENABLE_FTS4

- -DSQLITE_ENABLE_FTS5

La page source date du 2024-04-16, la traduction du 2024-04-24