Small. Fast. Reliable.
Choose any three.

Command Line Shell For SQLite

1. Getting Started

The SQLite project provides a simple command-line program named **sqlite3** (or **sqlite3.exe** on Windows) that allows the user to manually enter and execute SQL statements against an SQLite database. This document provides a brief introduction on how to use the **sqlite3** program.

Start the **sqlite3** program by typing "sqlite3" at the command prompt, optionally followed by the name the file that holds the SQLite database. If the named file does not exist, a new database file with the given name will be created automatically. If no database file is specified on the command-line, a temporary database is created, then deleted when the "sqlite3" program exits.

On startup, the **sqlite3** program will show a brief banner message then prompt you to enter SQL. Type in SQL statements (terminated by a semicolon), press "Enter" and the SQL will be executed.

For example, to create a new SQLite database named "ex1" with a single table named "tbl1", you might do this:

```
$ sqlite3 ex1
SQLite version 3.28.0 2019-03-02 15:25:24
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>
```

Terminate the sqlite3 program by typing your system End-Of-File character (usually a Control-D). Use the interrupt character (usually a Control-C) to stop a long-running SQL statement.

Make sure you type a semicolon at the end of each SQL command! The sqlite3 program looks for a semicolon to know when your SQL command is complete. If you omit the

semicolon, sqlite3 will give you a continuation prompt and wait for you to enter more text to be added to the current SQL command. This feature allows you to enter SQL commands that span multiple lines. For example:

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

2. Double-click Startup On Windows

Windows users can double-click on the **sqlite3.exe** icon to cause the command-line shell to pop-up a terminal window running SQLite. However, because double-clicking starts the sqlite3.exe without command-line arguments, no database file will have been specified, so SQLite will use a temporary database that is deleted when the session exits. To use a persistent disk file as the database, enter the ".open" command immediately after the terminal window starts up:

```
SQLite version 3.28.0 2019-03-02 15:25:24
Enter ".help" for usage hints.
Connected to a transient in-memory database.
Use ".open FILENAME" to reopen on a persistent database.
sqlite> .open ex1.db
sqlite>
```

The example above causes the database file named "ex1.db" to be opened and used. The "ex1.db" file is created if it does not previously exist. You might want to use a full pathname to ensure that the file is in the directory that you think it is in. Use forward-slashes as the directory separator character. In other words use "c:/work/ex1.db", not "c:\work\ex1.db".

Alternatively, you can create a new database using the default temporary storage, then save that database into a disk file using the ".save" command:

```
SQLite version 3.28.0 2019-03-02 15:25:24
Enter ".help" for usage hints.
Connected to a transient in-memory database.
Use ".open FILENAME" to reopen on a persistent database.
sqlite> ... many SQL commands omitted ...
sqlite> .save ex1.db
sqlite>
```

Be careful when using the ".save" command as it will overwrite any preexisting database files having the same name without prompting for confirmation. As with the ".open" command, you might want to use a full pathname with forward-slash directory separators to avoid ambiguity.

3. Special commands to sqlite3 (dot-commands)

Most of the time, sqlite3 just reads lines of input and passes them on to the SQLite library for execution. But input lines that begin with a dot (".") are intercepted and interpreted by the sqlite3 program itself. These "dot commands" are typically used to change the output format of queries, or to execute certain prepackaged query statements. There were originally just a few dot commands, but over the years many new features have accumulated so that today there over 60.

For a listing of the available dot commands, you can enter ".help" with no arguments. Or enter ".help TOPIC" for detailed information about TOPIC. The list of available dot-commands follows:

```
sqlite> .help
.archive ...             Manage SQL archives
.auth ON|OFF             Show authorizer callbacks
.backup ?DB? FILE        Backup DB (default "main") to FILE
.bail on|off             Stop after hitting an error.  Default OFF
.binary on|off           Turn binary output on or off.  Default OFF
.cd DIRECTORY            Change the working directory to DIRECTORY
.changes on|off          Show number of rows changed by SQL
.check GLOB              Fail if output since .testcase does not match
.clone NEWDB             Clone data into NEWDB from the existing database
.databases               List names and files of attached databases
.dbconfig ?op? ?val?     List or change sqlite3_db_config() options
.dbinfo ?DB?             Show status information about the database
.dump ?TABLE? ...        Render all database content as SQL
.echo on|off             Turn command echo on or off
.eqp on|off|full|...     Enable or disable automatic EXPLAIN QUERY PLAN
.excel                   Display the output of next command in a spreadsheet
.exit ?CODE?             Exit this program with return-code CODE
.expert                  EXPERIMENTAL. Suggest indexes for specified queries
.fullschema ?--indent?   Show schema and the content of sqlite_stat tables
.headers on|off          Turn display of headers on or off
.help ?-all? ?PATTERN?   Show help text for PATTERN
.import FILE TABLE       Import data from FILE into TABLE
.imposter INDEX TABLE    Create imposter table TABLE on index INDEX
.indexes ?TABLE?         Show names of indexes
.iotrace FILE            Enable I/O diagnostic logging to FILE
.limit ?LIMIT? ?VAL?     Display or change the value of an SQLITE_LIMIT
.lint OPTIONS            Report potential schema issues.
.load FILE ?ENTRY?       Load an extension library
.log FILE|off            Turn logging on or off.  FILE can be stderr/stdout
.mode MODE ?TABLE?       Set output mode
.nullvalue STRING        Use STRING in place of NULL values
.once (-e|-x|FILE)       Output for the next SQL command only to FILE
.open ?OPTIONS? ?FILE?   Close existing database and reopen FILE
.output ?FILE?           Send output to FILE or stdout if FILE is omitted
.parameter CMD ...       Manage SQL parameter bindings
.print STRING...         Print literal STRING
.progress N              Invoke progress handler after every N opcodes
.prompt MAIN CONTINUE    Replace the standard prompts
.quit                    Exit this program
.read FILE               Read input from FILE
.restore ?DB? FILE       Restore content of DB (default "main") from FILE
.save FILE               Write in-memory database into FILE
.scanstats on|off        Turn sqlite3_stmt_scanstatus() metrics on or off
```

```
.schema ?PATTERN?         Show the CREATE statements matching PATTERN
.selftest ?OPTIONS?       Run tests defined in the SELFTEST table
.separator COL ?ROW?      Change the column and row separators
.session ?NAME? CMD ...   Create or control sessions
.sha3sum ...              Compute a SHA3 hash of database content
.shell CMD ARGS...        Run CMD ARGS... in a system shell
.show                     Show the current values for various settings
.stats ?on|off?           Show stats or turn stats on or off
.system CMD ARGS...       Run CMD ARGS... in a system shell
.tables ?TABLE?           List names of tables matching LIKE pattern TABLE
.testcase NAME            Begin redirecting output to 'testcase-out.txt'
.timeout MS               Try opening locked tables for MS milliseconds
.timer on|off             Turn SQL timer on or off
.trace ?OPTIONS?          Output each SQL statement as it is run
.vfsinfo ?AUX?            Information about the top-level VFS
.vfslist                  List all available VFSes
.vfsname ?AUX?            Print the name of the VFS stack
.width NUM1 NUM2 ...      Set column widths for "column" mode
sqlite>
```

4. Rules for "dot-commands"

Ordinary SQL statements are free-form, and can be spread across multiple lines, and can have whitespace and comments anywhere. Dot-commands are more restrictive:

- A dot-command must begin with the "." at the left margin with no preceding whitespace.
- The dot-command must be entirely contained on a single input line.
- A dot-command cannot occur in the middle of an ordinary SQL statement. In other words, a dot-command cannot occur at a continuation prompt.
- Dot-commands do not recognize comments.

The dot-commands are interpreted by the sqlite3.exe command-line program, not by SQLite itself. So none of the dot-commands will work as an argument to SQLite interfaces like sqlite3_prepare() or sqlite3_exec().

5. Changing Output Formats

The sqlite3 program is able to show the results of a query in eight different formats: "csv", "column", "html", "insert", "line", "list", "quote", "tabs", and "tcl". You can use the ".mode" dot command to switch between these output formats.

The default output mode is "list". In list mode, each row of a query result is written on one line of output and each column within that row is separated by a specific separator string. The default separator is a pipe symbol ("|"). List mode is especially useful when you are going to send the output of a query to another program (such as AWK) for additional processing.

```
sqlite> .mode list
sqlite> select * from tbl1;
hello|10
goodbye|20
sqlite>
```

Use the ".separator" dot command to change the separator. For example, to change the separator to a comma and a space, you could do this:

```
sqlite> .separator ", "
sqlite> select * from tbl1;
hello, 10
goodbye, 20
sqlite>
```

The next ".mode" command will reset the ".separator" back to its default. So you will need repeat the ".separator" command whenever you change modes if you want to continue using a non-standard separator.

In "quote" mode, the output is formatted as SQL literals. Strings are enclosed in single-quotes and internal single-quotes are escaped by doubling. Blobs are displayed in hexadecimal blob literal notation (Ex: x'abcd'). Numbers are displayed as ASCII text and NULL values are shown as "NULL". All columns are separated from each other by a comma (or whatever alternative character is selected using ".separator").

```
sqlite> .mode quote
sqlite> select * from tbl1;
'hello',10
'goodbye',20
sqlite>
```

In "line" mode, each column in a row of the database is shown on a line by itself. Each line consists of the column name, an equal sign and the column data. Successive records are separated by a blank line. Here is an example of line mode output:

```
sqlite> .mode line
sqlite> select * from tbl1;
one = hello
two = 10

one = goodbye
two = 20
sqlite>
```

In column mode, each record is shown on a separate line with the data aligned in columns. For example:

```
sqlite> .mode column
sqlite> select * from tbl1;
one         two
----------  ----------
hello       10
goodbye     20
sqlite>
```

By default, each column is between 1 and 10 characters wide, depending on the column header name and the width of the first column of data. Data that is too wide to fit in a column is truncated. Use the ".width" dot-command to adjust column widths, like this:

```
sqlite> .width 12 6
```

```
sqlite> select * from tbl1;
one           two
------------  ------
hello         10
goodbye       20
sqlite>
```

The ".width" command in the example above sets the width of the first column to 12 and the width of the second column to 6. All other column widths were unaltered. You can gives as many arguments to ".width" as necessary to specify the widths of as many columns as are in your query results.

If you specify a column a width of 0, then the column width is automatically adjusted to be the maximum of three numbers: 10, the width of the header, and the width of the first row of data. This makes the column width self-adjusting. The default width setting for every column is this auto-adjusting 0 value.

Use a negative column width for right-justified columns.

The column labels that appear on the first two lines of output can be turned on and off using the ".header" dot command. In the examples above, the column labels are on. To turn them off you could do this:

```
sqlite> .header off
sqlite> select * from tbl1;
hello         10
goodbye       20
sqlite>
```

Another useful output mode is "insert". In insert mode, the output is formatted to look like SQL INSERT statements. Use insert mode to generate text that can later be used to input data into a different database.

When specifying insert mode, you have to give an extra argument which is the name of the table to be inserted into. For example:

```
sqlite> .mode insert new_table
sqlite> select * from tbl1;
INSERT INTO "new_table" VALUES('hello',10);
INSERT INTO "new_table" VALUES('goodbye',20);
sqlite>
```

The last output mode is "html". In this mode, sqlite3 writes the results of the query as an XHTML table. The beginning <TABLE> and the ending </TABLE> are not written, but all of the intervening <TR>s, <TH>s, and <TD>s are. The html output mode is envisioned as being useful for CGI.

6. Writing results to a file

By default, sqlite3 sends query results to standard output. You can change this using the ".output" and ".once" commands. Just put the name of an output file as an argument to .output and all subsequent query results will be written to that file. Or use the .once

command instead of .output and output will only be redirected for the single next command before reverting to the console. Use .output with no arguments to begin writing to standard output again. For example:

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

If the first character of the ".output" or ".once" filename is a pipe symbol ("|") then the remaining characters are treated as a command and the output is sent to that command. This makes it easy to pipe the results of a query into some other process. For example, the "open -f" command on a Mac opens a text editor to display the content that it reads from standard input. So to see the results of a query in a text editor, one could type:

```
sqlite3> .once '|open -f'
sqlite3> SELECT * FROM bigTable;
```

If the ".output" or ".once" commands have an argument of "-e" then output is collected into a temporary file and the system text editor is invoked on that text file. Thus, the command ".once -e" achieves the same result as ".once '|open -f'" but with the benefit of being portable across all systems.

If the ".output" or ".once" commands have a "-x" argument, that causes them to accumulate output as Comma-Separated-Values (CSV) in a temporary file, then invoke the default system utility for viewing CSV files (usually a spreadsheet program) on the result. This is a quick way of sending the result of a query to a spreadsheet for easy viewing:

```
sqlite3> .once -x
sqlite3> SELECT * FROM bigTable;
```

The ".excel" command is an alias for ".once -x". It does exactly the same thing.

6.1. File I/O Functions

The command-line shell adds two application-defined SQL functions that facilitate reading content from a file into a table column, and writing the content of a column into a file, respectively.

The readfile(X) SQL function reads the entire content of the file named X and returns that content as a BLOB. This can be used to load content into a table. For example:

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

The writefile(X,Y) SQL function write the blob Y into the file named X and returns the number of bytes written. Use this function to extract the content of a single table column into a file. For example:

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

Note that the readfile(X) and writefile(X,Y) functions are extension functions and are not built into the core SQLite library. These routines are available as a loadable extension in the ext/misc/fileio.c source file in the SQLite source code repositories.

6.2. The edit() SQL function

The CLI has another build-in SQL function named edit(). Edit() takes one or two arguments. The first argument is a value - usually a large multi-line string to be edited. The second argument is the name of a text editor. If the second argument is omitted, the VISUAL environment variable is used. The edit() function writes its first argument into a temporary file, invokes the editor on the temporary file, rereads the file back into memory after the editor is done, then returns the edited text.

The edit() function can be used to make changes to large text values. For example:

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

In this example, the content of the docs.body field for the entry where docs.name is "report-15" will be sent to the editor. After the editor returns, the result will be written back into the docs.body field.

The default operation of edit() is to invoke a text editor. But by using an alternative edit program in the second argument, you can also get it to edit images or other non-text resources. For example, if you want to modify a JPEG image that happens to be stored in a field of a table, you could run:

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

The edit program can also be used as a viewer, by simply ignoring the return value. For example, to merely look at the image above, you might run:

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

7. Querying the database schema

The sqlite3 program provides several convenience commands that are useful for looking at the schema of the database. There is nothing that these commands do that cannot be done by some other means. These commands are provided purely as a shortcut.

For example, to see a list of the tables in the database, you can enter ".tables".

```
sqlite> .tables
```

```
tbl1
tbl2
sqlite>
```

The ".tables" command is similar to setting list mode then executing the following query:

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

But the ".tables" command does more. It queries the sqlite_master table for all attached databases, not just the primary database. And it arranges its output into neat columns.

The ".indexes" command works in a similar way to list all of the indexes. If the ".indexes" command is given an argument which is the name of a table, then it shows just indexes on that table.

The ".schema" command shows the complete schema for the database, or for a single table if an optional tablename argument is provided:

```
sqlite> .schema
create table tbl1(one varchar(10), two smallint)
CREATE TABLE tbl2 (
  f1 varchar(30) primary key,
  f2 text,
  f3 real
)
sqlite> .schema tbl2
CREATE TABLE tbl2 (
  f1 varchar(30) primary key,
  f2 text,
  f3 real
)
sqlite>
```

The ".schema" command is roughly the same as setting list mode, then entering the following query:

```
SELECT sql FROM sqlite_master
ORDER BY tbl_name, type DESC, name
```

As with ".tables", the ".schema" command shows the schema for all attached databases. If you only want to see the schema for a single database (perhaps "main") then you can add an argument to ".schema" to restrict its output:

```
sqlite> .schema main.*
```

The ".schema" command can be augmented with the "--indent" option, in which case it tries to reformat the various CREATE statements of the schema so that they are more easily readable by humans.

The ".databases" command shows a list of all databases open in the current connection. There will always be at least 2. The first one is "main", the original database opened. The second is "temp", the database used for temporary tables. There may be additional

databases listed for databases attached using the ATTACH statement. The first output column is the name the database is attached with, and the second column is the filename of the external file.

```
sqlite> .databases
```

The ".fullschema" dot-command works like the ".schema" command in that it displays the entire database schema. But ".fullschema" also includes dumps of the statistics tables "sqlite_stat1", "sqlite_stat3", and "sqlite_stat4", if they exist. The ".fullschema" command normally provides all of the information needed to exactly recreate a query plan for a specific query. When reporting suspected problems with the SQLite query planner to the SQLite development team, developers are requested to provide the complete ".fullschema" output as part of the trouble report. Note that the sqlite_stat3 and sqlite_stat4 tables contain samples of index entries and so might contain sensitive data, so do not send the ".fullschema" output of a proprietary database over a public channel.

8. CSV Import

Use the ".import" command to import CSV (comma separated value) data into an SQLite table. The ".import" command takes two arguments which are the name of the disk file from which CSV data is to be read and the name of the SQLite table into which the CSV data is to be inserted.

Note that it is important to set the "mode" to "csv" before running the ".import" command. This is necessary to prevent the command-line shell from trying to interpret the input file text as some other format.

```
sqlite> .mode csv
sqlite> .import C:/work/somedata.csv tab1
```

There are two cases to consider: (1) Table "tab1" does not previously exist and (2) table "tab1" does already exist.

In the first case, when the table does not previously exist, the table is automatically created and the content of the first row of the input CSV file is used to determine the name of all the columns in the table. In other words, if the table does not previously exist, the first row of the CSV file is interpreted to be column names and the actual data starts on the second row of the CSV file.

For the second case, when the table already exists, every row of the CSV file, including the first row, is assumed to be actual content. If the CSV file contains an initial row of column labels, that row will be read as data and inserted into the table. To avoid this, make sure that table does not previously exist.

9. CSV Export

To export an SQLite table (or part of a table) as CSV, simply set the "mode" to "csv" and then run a query to extract the desired rows of the table.

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

In the example above, the ".header on" line causes column labels to be printed as the first row of output. This means that the first row of the resulting CSV file will contain column labels. If column labels are not desired, set ".header off" instead. (The ".header off" setting is the default and can be omitted if the headers have not been previously turned on.)

The line ".once *FILENAME*" causes all query output to go into the named file instead of being printed on the console. In the example above, that line causes the CSV content to be written into a file named "C:/work/dataout.csv".

The final line of the example (the ".system c:/work/dataout.csv") has the same effect as double-clicking on the c:/work/dataout.csv file in windows. This will typically bring up a spreadsheet program to display the CSV file.

That command only works as written on Windows. The equivalent line on a Mac would be:

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

On Linux and other unix systems you will need to enter something like:

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

9.1. Export to Excel

To simplify export to a spreadsheet, the CLI provides the ".excel" command which captures the output of a single query and sends that output to the default spreadsheet program on the host computer. Use it like this:

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

The command above writes the output of the query as CSV into a temporary file, invokes the default handler for CSV files (usually the preferred spreadsheet program such as Excel or LibreOffice), then deletes the temporary file. This is essentially a short-hand method of doing the sequence of ".csv", ".once", and ".system" commands described above.

The ".excel" command is really an alias for ".once -x". The -x option to .once causes it to writes results as CSV into a temporary file that is named with a ".csv" suffix, then invoke the systems default handler for CSV files.

There is also a ".once -e" command which works similarly, except that it names the temporary file with a ".txt" suffix so that the default text editor for the system will be invoked, instead of the default spreadsheet.

10. Converting An Entire Database To An ASCII Text File

Use the ".dump" command to convert the entire contents of a database into a single ASCII text file. This file can be converted back into a database by piping it back into **sqlite3**.

A good way to make an archival copy of a database is this:

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

This generates a file named **ex1.dump.gz** that contains everything you need to reconstruct the database at a later time, or on another machine. To reconstruct the database, just type:

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

The text format is pure SQL so you can also use the .dump command to export an SQLite database into other popular SQL database engines. Like this:

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

11. Recover Data From a Corrupted Database

Like the ".dump" command, ".recover" attempts to convert the entire contents of a database file to text. The difference is that instead of reading data using the normal SQL database interface, ".recover" attempts to reassemble the database based on data extracted directly from as many database pages as possible. If the database is corrupt, ".recover" is usually able to recover data from all uncorrupted parts of the database, whereas ".dump" stops when the first sign of corruption is encountered.

If the ".recover" command recovers one or more rows that it cannot attribute to any database table, the output script creates a "lost_and_found" table to store the orphaned rows. The schema of the lost_and_found table is as follows:

```
CREATE TABLE lost_and_found(
    rootpgno INTEGER,            -- root page of tree pgno is a part of
    pgno INTEGER,                -- page number row was found on
    nfield INTEGER,              -- number of fields in row
    id INTEGER,                  -- value of rowid field, or NULL
```

```
    c0, c1, c2, c3...             -- columns for fields of row
);
```

The "lost_and_found" table contains one row for each orphaned row recovered from the database. Additionally, there is one row for each recovered index entry that cannot be attributed to any SQL index. This is because, in an SQLite database, the same format is used to store SQL index entries and WITHOUT ROWID table entries.

Column	Contents
rootpgno	Even though it may not be possible to attribute the row to a specific database table, it may be part of a tree structure within the database file. In this case, the root page number of that tree structure is stored in this column. Or, if the page the row was found on is not part of a tree structure, this column stores a copy of the value in column "pgno" - the page number of the page the row was found on. In many, although not all, cases, all rows in the lost_and_found table with the same value in this column belong to the same table.
pgno	The page number of the page on which this row was found.
nfield	The number of fields in this row.
id	If the row comes from a WITHOUT ROWID table, this column contains NULL. Otherwise, it contains the 64-bit integer rowid value for the row.
c0, c1, c2...	The values for each column of the row are stored in these columns. The ".recover" command creates the lost_and_found table with as many columns as required by the longest orphaned row.

If the recovered database schema already contains a table named "lost_and_found", the ".recover" command uses the name "lost_and_found0". If the name "lost_and_found0" is also already taken, "lost_and_found1", and so on. The default name "lost_and_found" may be overridden by invoking ".recover" with the --lost-and-found switch. For example, to have the output script call the table "orphaned_rows":

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

12. Loading Extensions

You can add new custom application-defined SQL functions, collating sequences, virtual tables, and VFSes to the command-line shell at run-time using the ".load" command. First, convert the extension in to a DLL or shared library (as described in the Run-Time Loadable Extensions document) then type:

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

Note that SQLite automatically adds the appropriate extension suffix (".dll" on windows, ".dylib" on Mac, ".so" on most other unixes) to the extension filename. It is generally a good idea to specify the full pathname of the extension.

SQLite computes the entry point for the extension based on the extension filename. To override this choice, simply add the name of the extension as a second argument to the ".load" command.

Source code for several useful extensions can be found in the ext/misc subdirectory of the SQLite source tree. You can use these extensions as-is, or as a basis for creating your own custom extensions to address your own particular needs.

13. Cryptographic Hashes Of Database Content

The ".sha3sum" dot-command computes a SHA3 hash of the *content* of the database. To be clear, the hash is computed over the database content, not its representation on disk. This means, for example, that a VACUUM or similar data-preserving transformation does not change the hash.

The ".sha3sum" command supports options "--sha3-224", "--sha3-256", "--sha3-384", and "--sha3-512" to define which variety of SHA3 to use for the hash. The default is SHA3-256.

The database schema (in the sqlite_master table) is not normally included in the hash, but can be added by the "--schema" option.

The ".sha3sum" command takes a single optional argument which is a LIKE pattern. If this option is present, only tables whose names match the LIKE pattern will be hashed.

The ".sha3sum" command is implemented with the help of the extension function "sha3_query()" that is included with the command-line shell.

14. Database Content Self-Tests

The ".selftest" command attempts to verify that a database is intact and is not corrupt. The .selftest command looks for a table in schema named "selftest" and defined as follows:

```
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
);
```

The .selftest command reads the rows of the selftest table in selftest.tno order. For each 'memo' row, it writes the text in 'cmd' to the output. For each 'run' row, it runs the 'cmd'

text as SQL and compares the result to the value in 'ans', and shows an error message if the results differ.

If there is no selftest table, the ".selftest" command runs PRAGMA integrity_check.

The ".selftest --init" command creates the selftest table if it does not already exists, then appends entries that check the SHA3 hash of the content of all tables. Subsequent runs of ".selftest" will verify that the database has not been changed in any way. To generates tests to verify that a subset of the tables are unchanged, simply run ".selftest --init" then DELETE the selftest rows that refer to tables that are not constant.

15. SQLite Archive Support

The ".archive" dot-command and the "-A" command-line option provide built-in support for the SQLite Archive format. The interface is similar to that of the "tar" command on unix systems. Each invocation of the ".ar" command must specify a single command option. The following commands are available for ".archive":

Option	Long Option	Purpose
-c	--create	Create a new archive containing specified files.
-x	--extract	Extract specified files from archive.
-i	--insert	Add files to existing archive.
-t	--list	List the files in the archive.
-u	--update	Add files to existing archive *if* they have changed.

As well as the command option, each invocation of ".ar" may specify one or more modifier options. Some modifier options require an argument, some do not. The following modifier options are available:

Option	Long Option	Purpose
-v	--verbose	List each file as it is processed.
-f FILE	--file FILE	If specified, use file FILE as the archive. Otherwise, assume that the current "main" database is the archive to be operated on.
-a FILE	--append FILE	Like --file, use file FILE as the archive, but open the file using the apndvfs VFS so that the archive will be appended to the end of FILE if FILE already exists.
-C DIR	--directory DIR	If specified, interpret all relative paths as relative to DIR, instead of the current working directory.
-n	--dryrun	Show the SQL that would be run

		to carry out the archive operation, but do not actually change anything.
--	--	All subsequent command line words are command arguments, not options.

For command-line usage, add the short style command-line options immediately following the "-A", without an intervening space. All subsequent arguments are considered to be part of the .archive command. For example, the following commands are equivalent:

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

Long and short style options may be mixed. For example, the following are equivalent:

```
-- Two ways to create a new archive named "new_archive.db" containing
-- files "file1", "file2" and "file3".
.ar -c --file new_archive.db file1 file2 file3
.ar -f new_archive.db --create file1 file2 file3
```

Alternatively, the first argument following to ".ar" may be the concatenation of the short form of all required options (without the "-" characters). In this case arguments for options requiring them are read from the command line next, and any remaining words are considered command arguments. For example:

```
-- Create a new archive "new_archive.db" containing files "file1" and
-- "file2" from directory "dir1".
.ar cCf dir1 new_archive.db file1 file2 file3
```

15.1. SQLite Archive Create Command

Create a new archive, overwriting any existing archive (either in the current "main" db or in the file specified by a --file option). Each argument following the options is a file to add to the archive. Directories are imported recursively. See above for examples.

15.2. SQLite Archive Extract Command

Extract files from the archive (either to the current working directory or to the directory specified by a --directory option). If there are no arguments following the options all files are extracted from the archive. Or, if there are arguments, they are the names of files to extract from the archive. Any specified directories are extracted recursively. It is an error if any specified files are not part of the archive.

```
-- Extract all files from the archive in the current "main" db to the
-- current working directory. List files as they are extracted.
.ar --extract --verbose
```

```
-- Extract file "file1" from archive "ar.db" to directory "dir1".
.ar fCx ar.db dir1 file1
```

15.3. SQLite Archive List Command

List the contents of the archive. If no arguments are specified, then all files are listed. Otherwise, only those specified as arguments are. Currently, the --verbose option does not change the behaviour of this command. That may change in the future.

```
-- List contents of archive in current "main" db..
.ar --list
```

15.4. SQLite Archive Insert And Update Commands

The --update and --insert commands work like --create command, except that they do not delete the current archive before commencing. New versions of files silently replace existing files with the same names, but otherwise the initial contents of the archive (if any) remain intact.

For the --insert command, all files listed are inserted into the archive. For the --update command, files are only inserted if they do not previously exist in the archive, or if their "mtime" or "mode" is different from what is currently in the archive.

Compatibility node: Prior to SQLite version 3.28.0 (2019-04-16) only the --update option was supported but that option worked like --insert in that it always reinserted every file regardless of whether or not it had changed.

15.5. Operations On ZIP Archives

If FILE is a ZIP archive rather than an SQLite Archive, the ".archive" command and the "-A" command-line option still work. This is accomplished using of the zipfile extension. Hence, the following commands are roughly equivalent, differing only in output formatting:

Traditional Command	Equivalent sqlite3.exe Command
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

15.6. SQL Used To Implement SQLite Archive Operations

The various SQLite Archive Archive commands are implemented using SQL statements. Application developers can easily add SQLite Archive Archive reading and writing support to their own projects by running the appropriate SQL.

To see what SQL statements are used to implement an SQLite Archive operation, add the --dryrun or -n option. This causes the SQL to be displayed but inhibits the execution of

the SQL.

The SQL statements used to implement SQLite Archive operations make use of various loadable extensions. These extensions are all available in the SQLite source tree in the ext/misc/ subfolder. The extensions needed for full SQLite Archive support include:

1. fileio.c — This extension adds SQL functions readfile() and writefile() for reading and writing content from files on disk. The fileio.c extension also includes fsdir() table-valued function for listing the contents of a directory and the lsname() function for converting numeric st_mode integers from the stat() system call into human-readable strings after the fashion of the "ls -l" command.

2. sqlar.c — This extension adds the sqlar_compress() and sqlar_uncompress() functions that are needed to compress and uncompress file content as it is insert and extracted from an SQLite Archive.

3. zipfile.c — This extension implements the "zipfile(FILE)" table-valued function which is used to read ZIP archives. This extension is only needed when reading ZIP archives instead of SQLite archives.

4. appendvfs.c — This extension implements a new VFS that allows an SQLite database to be appended to some other file, such as an executable. This extension is only needed if the --append option to the .archive command is used.

16. SQL Parameters

SQLite allows bound parameters to appear in an SQL statement anywhere that a literal value is allowed. The values for these parameters are set using the sqlite3_bind_...() family of APIs.

Parameters can be either named or unnamed. An unnamed parameter is a single question mark ("?"). Named parameters are a "?" followed immediately by a number (ex: "?15" or "?123") or one of the characters "$", ":", or "@" followed by an alphanumeric name (ex: "$var1", ":xyz", "@bingo").

This command-line shell leaves unnamed parameters unbound, meaning that they will have a value of an SQL NULL, but named parameters might be assigned values. If there exists a TEMP table named "sqlite_parameters" with a schema like this:

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

And if there is an entry in that table where the key column exactly matches the name of parameter (including the initial "?", "$", ":", or "@" character) then the parameter is assigned the value of the value column. If no entry exists, the parameter defaults to NULL.

The ".parameter" command exists to simplify managing this table. The ".parameter init" command (often abbreviated as just ".param init") creates the temp.sqlite_parameters

table if it does not already exist. The ".param list" command shows all entries in the temp.sqlite_parameters table. The ".param clear" command drops the temp.sqlite_parameters table. The ".param set KEY VALUE" and ".param unset KEY" commands create or delete entries from the temp.sqlite_parameters table.

The temp.sqlite_parameters table only provides values for parameters in the command-line shell. The temp.sqlite_parameter table has no affect on queries that are run directly using the SQLite C-language API. Individual applications are expected to implement their own parameter binding. You can search for "sqlite_parameters" in the command-line shell source code to see how the command-line shell does parameter binding, and use that as a hint for how to implement it yourself.

17. Index Recommendations (SQLite Expert)

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

For most non-trivial SQL databases, the key to performance is creating the right SQL indexes. In this context "the right SQL indexes" means those that cause the queries that an application needs to optimize run fast. The ".expert" command can assist with this by proposing indexes that might assist with specific queries, were they present in the database.

The ".expert" command is issued first, followed by the SQL query on a separate line. For example, consider the following session:

```
sqlite> CREATE TABLE x1(a, b, c);                   -- Create table in database
sqlite> .expert
sqlite> SELECT * FROM x1 WHERE a=? AND b>?;         -- Analyze this SELECT
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);              -- Create the recommended index
sqlite> .expert
sqlite> SELECT * FROM x1 WHERE a=? AND b>?;         -- Re-analyze the same SELECT
(no new indexes)

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

In the above, the user creates the database schema (a single table - "x1"), and then uses the ".expert" command to analyze a query, in this case "SELECT * FROM x1 WHERE a=? AND b>?". The shell tool recommends that the user create a new index (index "x1_idx_000123a7") and outputs the plan that the query would use in EXPLAIN QUERY PLAN format. The user then creates an index with an equivalent schema and runs the analysis on the same query again. This time the shell tool does not recommend any new indexes, and outputs the plan that SQLite will use for the query given the existing indexes.

The ".expert" command accepts the following options:

Option	Purpose

--verbose	If present, output a more verbose report for each query analyzed.
--sample PERCENT	By default, the ".expert" command recommends indexes based on the query and database schema alone. This is similar to the way the SQLite query planner selects indexes for queries if the user has not run the ANALYZE command on the database to generate data distribution statistics. If this option is passed a non-zero argument, the ".expert" command generates similar data distribution statistics for all indexes considered based on PERCENT percent of the rows currently stored in each database table. For databases with unusual data distributions, this may lead to better index recommendations, particularly if the application intends to run ANALYZE. For small databases and modern CPUs, there is usually no reason not to pass "--sample 100". However, gathering data distribution statistics can be expensive for large database tables. If the operation is too slow, try passing a smaller value for the --sample option.

Th functionality described in this section may be integrated into other applications or tools using the SQLite expert extension code.

18. Other Dot Commands

There are many other dot-commands available in the command-line shell. See the ".help" command for a complete list for any particular version and build of SQLite.

19. Using sqlite3 in a shell script

One way to use sqlite3 in a shell script is to use "echo" or "cat" to generate a sequence of commands in a file, then invoke sqlite3 while redirecting input from the generated command file. This works fine and is appropriate in many circumstances. But as an added convenience, sqlite3 allows a single SQL command to be entered on the command line as a second argument after the database name. When the sqlite3 program is launched with two arguments, the second argument is passed to the SQLite library for processing, the query results are printed on standard output in list mode, and the program exits. This mechanism is designed to make sqlite3 easy to use in conjunction with programs like "awk". For example:

```
$ 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
$
```

20. Ending shell commands

SQLite commands are normally terminated by a semicolon. In the shell you can also use the word "GO" (case-insensitive) or a slash character "/" on a line by itself to end a command. These are used by SQL Server and Oracle, respectively. These won't work in **sqlite3_exec()**, because the shell translates these into a semicolon before passing them to that function.

21. Compiling the sqlite3 program from sources

To compile the command-line shell on unix systems and on Windows with MinGW, the usual configure-make command works:

```
sh configure; make
```

The configure-make works whether your are building from the canonical sources from the source tree, or from an amalgamated bundle. There are few dependencies. When building from canonical sources, a working tclsh is required. If using an amalgamation bundle, all the preprocessing work normally done by tclsh will have already been carried out and only normal build tools are required.

A working zlib compression library is needed in order for the .archive command to operate.

On Windows with MSVC, use nmake with the Makefile.msc:

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

For correct operation of the .archive command, make a copy of the zlib source code into the compat/zlib subdirectory of the source tree and compile this way:

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

21.1. Do-It-Yourself Builds

The source code to the sqlite3 command line interface is in a single file named "shell.c". The shell.c source file is generated from other sources, but most of the code for shell.c can be found in src/shell.c.in. (Regenerate shell.c by typing "make shell.c" from the canonical source tree.) Compile the shell.c file (together with the sqlite3 library source code) to generate the executable. For example:

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

The following additional compile-time options are recommended in order to provide a full-featured command-line shell:

- -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