Hints for Debugging SQLite

sqlite.org/debugging.html

Small. Fast. Reliable.
Choose any three.

Debugging Hints

The following is a random assortment of techniques used by the SQLite developers to trace, examine, and understand the behavior of the core SQLite library.

These techniques are designed to aid in understanding the core SQLite library itself, not applications that merely use SQLite.

1. **Use the ".eqp full" option on the command-line shell**

   When you have a SQL script that you are debugging or trying to understand, it is often useful to run it in the command-line shell with the ".eqp full" setting. When ".eqp" is set to FULL, the shell automatically shows the EXPLAIN and EXPLAIN QUERY PLAN output for each command prior to actually running that command.

   For added readability, also set ".echo on" so that the output contains the original SQL text.

   The newer ".eqp trace" command does everything that ".eqp full" does and also turns on VDBE tracing.

2. **Use compile-time options to enable debugging features.**

   Suggested compile-time options include:

   - -DSQLITE_DEBUG
   - -DSQLITE_ENABLE_EXPLAIN_COMMENTS
   - -DSQLITE_ENABLE_TREETRACE
   - -DSQLITE_ENABLE_WHERETRACE

   The SQLITE_ENABLE_TREETRACE and SQLITE_ENABLE_WHERETRACE options are not documented in compile-time options document because they are not officially supported. What they do is activate the ".treetrace" and ".wheretrace" dot-commands in the command-line shell, which provide low-level tracing output for the logic that generates code for SELECT and DML statements and WHERE clauses, respectively.

3. **Call sqlite3ShowExpr() and similar from the debugger.**

   When compiled with SQLITE_DEBUG, SQLite includes routines that will print out various internal abstract syntax tree structures as ASCII-art graphs. This can be very useful in a debugging in order to understand the variables that SQLite is working with. The following routines are available:

   - void sqlite3ShowExpr(const Expr*);
   - void sqlite3ShowExprList(const ExprList*);
   - void sqlite3ShowIdList(const IdList*);
   - void sqlite3ShowSrcList(const SrcList*);
   - void sqlite3ShowSelect(const Select*);
   - void sqlite3ShowWith(const With*);
   - void sqlite3ShowUpsert(const Upsert*);
   - void sqlite3ShowTrigger(const Trigger*);
   - void sqlite3ShowTriggerList(const Trigger*);
   - void sqlite3ShowTriggerStep(const TriggerStep*);
   - void sqlite3ShowTriggerStepList(const TriggerStep*);
   - void sqlite3ShowWindow(const Window*);
   - void sqlite3ShowWinFunc(const Window*);

   These routines are not APIs and are subject to change. They are for interactive debugging use only.

4. **Breakpoints on test_addoptrace**

   When debugging the bytecode generator, it is often useful to know where a particular opcode is being generated. To find this easily, run the script in a debugger. Set a breakpoint on the "test_addoptrace" routine. Then run the "PRAGMA vdbe_addoptrace=ON;" followed by the SQL statement in question. Each opcode will be displayed as it is appended to the VDBE program, and the breakpoint will fire immediately thereafter. Step until reaching the opcode then look backwards in the stack to see where and how it was generated.

   This only works when compiled with SQLITE_DEBUG.

5. **Using the ".treetrace" and ".wheretrace" shell commands**

   When the command-line shell and the core SQLite library are both compiled with SQLITE_DEBUG and SQLITE_ENABLE_TREETRACE and SQLITE_ENABLE_WHERETRACE, then the shell has two commands used to turn on debugging facilities for the most intricate parts of the code generator - the logic dealing with SELECT statements and WHERE clauses, respectively. The ".treetrace" and ".wheretrace" commands each take a numeric argument which can be expressed in hexadecimal. Each bit turns on various parts of debugging. Values of "0xfff" and "0xff" are commonly used. Use an argument of "0" to turn all tracing output back off.

6. **Using the ".breakpoint" shell command**

   The ".breakpoint" command in the CLI does nothing but invoke the procedure named "test_breakpoint()", which is a no-op.

   If you have a script and you want to start debugging at some point half-way through that script, simply set a breakpoint in gdb (or whatever debugger you are using) on the test_breakpoint() function, and add a ".breakpoint" command where you want to stop. When you reach that first breakpoint, set whatever additional breakpoints are variable traces you need.

7. **Disable the lookaside memory allocator**

   When looking for memory allocation problems (memory leaks, use-after-free errors, buffer overflows, etc) it is sometimes useful to disable the lookaside memory allocator then run the test under valgrind or MSAN or some other heap memory debugging tool. The lookaside memory allocator can be disabled at start-time using the SQLITE_CONFIG_LOOKASIDE interface. The command-line shell will use that interface to disable lookaside if it is started with the "--lookaside 0 0" command line option.