Specializing SQLite's bytecode interpreter with proved control transfers

Anonymous · 23 November 2026

Abstract. We specialized SQLite 3.54.1 on eight prepared statements. With native GCC optimization, LTO and profile training, the residual interpreter reduced elapsed time by 10.7–11.2% across two paired runs on two fixture sizes. We give the conditions for resolving control transfers, compare stock and earlier emitters, and supply the source, bytecode snapshots and raw timings.

A prepared program as an input

A SQLite statement returns a row by suspending its virtual database engine (VDBE). The next call to `sqlite3_step` resumes the interpreter. On a table scan, the same short sequence of instructions can run thousands of times. Preparing the SQL determines many register numbers and jump destinations before any rows are read. We asked whether specializing the interpreter on those facts could reduce the elapsed time of an entire execution, including its returns to the caller. SQLite describes this execution model in its VDBE documentation [1].

Offline partial evaluation separates values known during specialization from values supplied at execution time. Tempo applied that method to systems C [2]. Following work on flow-sensitive binding-time analysis [3], our evaluator distinguishes program points where a value is known from points reached after a call or store that could change it. Earlier work on interpreter dispatch and on optimizing the boundary between SQLite and its host language motivates the performance question [4, 5]. We emit C from the interpreter itself and use an ordinary optimizing C compiler for machine code generation.

Which transfers can be resolved

The input consists of the preprocessed interpreter and a snapshot of the prepared statement's instruction array. A Rust C front end builds a scoped control-flow graph. For each eligible program counter (PC), the evaluator projects the selected path through the original opcode switch. It folds operand loads and branch conditions only when their C types, values and relevant effects permit substitution. A call can invalidate memory facts while leaving an unescaped automatic variable intact. A store through an unknown alias also invalidates affected facts.

The caller must choose the variant for a live program that matches the snapshot's immutable fields and C layout. The compiled code checks the current opcode at entry to a projected body. Unexpected opcodes use the original switch. Frame, program length and bytecode-base checks protect dispatch when the evaluator cannot establish that the corresponding facts survived. Mutable `Once` and initialization operands stay live. Four of the 211 instruction positions across the eight programs contain self-modifying `AggStep` or `String8` instructions and use the original dispatcher. A length or opcode match alone does not certify the other operands. A production integration must invalidate the variant when SQLite recompiles the statement.

The earlier emitter remembered the previous PC and consulted candidate destinations before returning to a general PC switch. A hit still incurred selection work. The revision removes this history. When the original switch exit proves the cursor equals an element of the current bytecode array, the emitter sends control directly to the proven successor. It recognizes the original loop increment, constant loop test and iteration prelude. Other loop forms retain ordinary control flow.

For example, the mixed-type query has a comparison at PC 6 and a jump destination of 9. On an early path to the shared jump label, analysis proves that the original assignment sets the cursor to element 8. The loop increment therefore sets it to element 9. The residual code sets the cursor to that element, executes the original step-count prelude, and transfers to the checked body for PC 9. On a later path through the same label, a helper may change the operand. That path still uses the live value. Combining both paths before analysis would discard the proof available on the first.

A second reduction applies when the target is dynamic. If the original assignment computes a cursor from an integer index, the emitter can retain that evaluated integer. It requires all paths to the exit to preserve the relationship between cursor and base. It performs the pointer assignment and observes the branch, progress callback and interrupt behavior at their original program points. Dispatch can then use the saved index directly. This removes a round trip from integer to pointer and back. The same reasoning applies at a recognized resume entry.

For these two reductions, equivalence follows one iteration at a time. Both executions arrive at dispatch

with the same live cursor and program state. For a direct edge, the proven array offset and original increment establish the next cursor. For a dynamic edge, the next index comes from the original evaluation, including its integer conversion. Executing the original prelude exactly once preserves the step count. If a proof fails, the original operation remains available. These arguments specify the local proof obligations. The implementation has differential tests, and the full evaluator has no machine-checked correctness proof.

Workloads and controls

We used the public SQLite 3.54.1 amalgamation, release 3540100. The eight SQL statements exercise a filtered arithmetic aggregate (Q0), an indexed join (Q1), grouped aggregation with sorting and a limit (Q2), a recursive coroutine (Q3), a mixed-type `CASE` expression (Q4), a parameterized scan (Q5), a primary-key lookup (Q6), and an uncorrelated `EXISTS` subquery (Q7). Two in-memory fixtures contained 8 or 4,096 rows in the main table. The coroutine generates eight values regardless of table size. Both fixtures produced identical bytecode snapshots.

The correctness harness compared row order, SQLite storage class, integer values, hexadecimal floating-point output, text and blob bytes, terminal return codes, and VM-step counts. Each statement ran three times with reset. The two parameterized queries cycled through bindings of 2, 6 and 10. An interrupt during the coroutine exercised its error exit. A separate check observed zero automatic recompilations over 480 executions and 120 parameter bindings. Its schema-change control observed one recompile on each fixture. These counters use SQLite's public statement-status interface [6].

We compared three arms. *Plain stock* was a separate executable containing the original interpreter. *Matched stock* and *specialized* shared a second executable containing the original interpreter and all eight generated functions. After warm-up, both used the same statement-identity check and indirect call. The measurement verified which interpreter ran and checked the result-row and VM-step totals for each timed batch. The two controls expose how linkage, code placement and the selection call can affect a comparison.

GCC 13.3 compiled both executables as GNU C17 with `-O3 -march=native -flto` and `-DNDEBUG`. The primary configuration additionally used profile-guided optimization (PGO). Both instrumented executables trained on the same sequence of queries and sizes twice, with 192 statement executions per case after warm-up. The linked executable used each interpreter in one pass. GCC rebuilt both binaries with the collected profiles before measurement. We also measured the same source without profile training. GCC's `-Ofast` permits transformations outside standard floating-point semantics and was outside the experiment [7].

All measured builds for the revised comparison set `_FORTIFY_SOURCE=0`. With fortified libc, a GCC object-size query can expose knowledge introduced by specialization [8]. In one control, substituting a known index changed a fortified call on an invalid file descriptor from returning an error to aborting before the call. The evaluator diagnoses observable compiler-knowledge queries in unsupported inputs. Ordinary libc, a pinned LP64 target, and the matching-snapshot requirement delimit the experiment.

On an Intel Core i7-10875H running Ubuntu 24.04 under WSL2, each batch ran in a new process pinned to guest CPU 8. Timing began after database setup, statement preparation and three warm-ups. It covered stepping, counting rows, reset, rebinding and reading the VM-step counter in bounded batches. Each of 18 rounds ran all 16 query/fixture pairs in a seeded order. Each pair used all six orders of the three arms equally often. Two scheduling seeds supplied a primary run and a replication.

We divide baseline time by specialized time within a round. Ratios above one favor specialization. For the full mix, we first average one execution of each of the 16 workloads within each arm, then divide the two means. This weights executions equally and gives longer queries a larger share of elapsed time. The reported quartiles describe round-to-round variation.

Results

With PGO, plain SQLite took 1.126 and 1.119 times as long as the revised interpreter on the full mix. Expressed as a reduction in elapsed time, the two estimates are 11.2% and 10.7%. The revision won all 18 paired rounds in each run. The matched-stock comparisons gave ratios of 1.127 and 1.110. On the small fixture alone the ratios against plain stock were 1.035 and 1.031. On the large fixture they were 1.126 and 1.120.

Table 1: Profile-guided primary run. Stock and specialized columns give median microseconds per complete execution. Each ratio is the median of paired times within rounds. Fixture sizes refer to rows in the main table.

	8 rows			4,096 rows		
Query	Stock	Spec.	Ratio	Stock	Spec.	Ratio
Q0	0.597	0.523	1.130	164.743	127.336	1.325
Q1	0.679	0.683	0.991	456.964	442.427	1.041
Q2	2.946	2.889	1.024	425.390	369.730	1.133
Q3	2.494	2.433	1.022	2.525	2.475	1.022
Q4	0.683	0.579	1.157	219.400	166.910	1.333
Q5	0.518	0.498	1.046	192.079	181.626	1.057
Q6	0.245	0.241	1.017	0.284	0.279	1.002
Q7	1.118	1.136	0.990	312.684	282.133	1.100

The largest repeatable gains came from the large-fixture arithmetic aggregate and mixed-type expression. Q0 took 165 µs in stock and 127 µs after specialization in the primary run. Q4 fell from 219 µs to 167 µs. The primary-key lookup on that fixture was within roughly one percent of parity in both runs. Some small-fixture medians also changed sides between runs. Figure 1 shows the spread for each case.

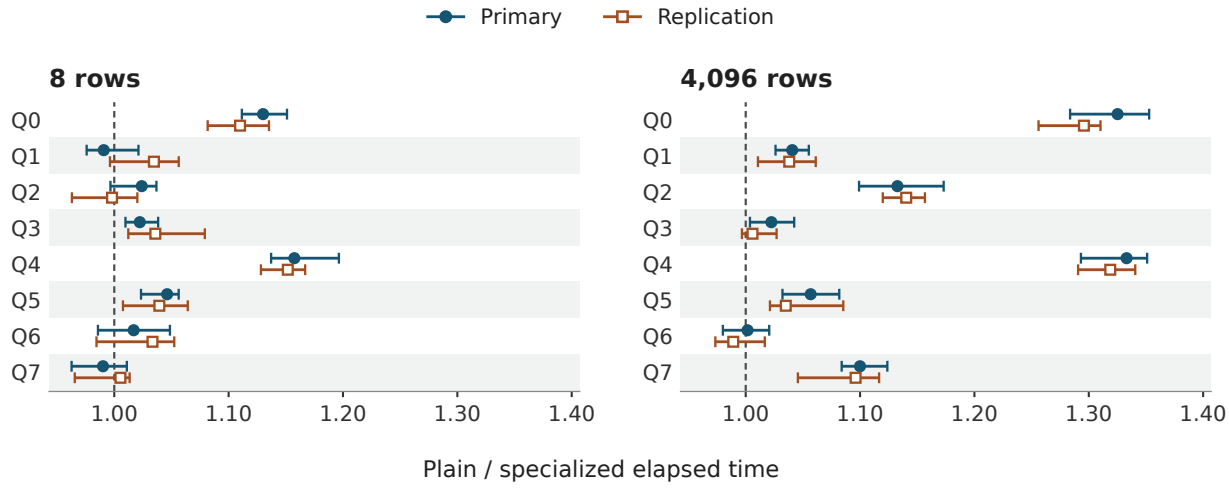

Figure 1: Paired stock-to-specialized time ratios for the profile-guided builds. Points mark medians and bars span the middle half of the 18 paired rounds. Values to the right of one favor specialization.

Table 2: Equal-execution results across all 16 workloads. The middle half and win count refer to plain stock. The matched column uses the original interpreter linked into the specialized executable. Native builds use GCC optimization and LTO without profiles. PGO adds matched training.

Emitter and compiler	Run	Plain ratio	Middle half	Wins	Matched ratio
Earlier, native	1	0.898	0.889–0.913	0/18	0.898
Revised, native	1	1.009	0.991–1.016	11/18	1.009
Revised, native	2	1.010	0.987–1.023	11/18	1.018
Earlier, PGO	1	1.010	1.001–1.039	14/18	1.015
Earlier, PGO	2	1.020	1.008–1.037	15/18	1.015
Revised, PGO	1	1.126	1.121–1.137	18/18	1.127
Revised, PGO	2	1.119	1.107–1.131	18/18	1.110

Under ordinary native optimization the earlier emitter took about 11.4% longer than stock. Its full-workload ratio was 0.898 and it lost every paired round. The revision brought the two unprofiled medians to 1.009 and 1.010. Both interquartile ranges crossed parity. On the eight-row fixture alone, ordinary stock remained faster. With profile training, the earlier emitter reached ratios of 1.010 and 1.020. The revised emitter reached 1.126 and 1.119 under that same training procedure. Absolute plain-stock times drifted between sessions, including two historical PGO runs on the same stock binary (148 and 118 µs

for the full mix). Each comparison therefore uses the stock arm from its own round.

Interpretation and limits

Specialization saves time only when the machine instructions it removes outweigh the work used to select and execute a residual path. The earlier emitter carried previous-PC state, checked candidate targets, and could then repeat general dispatch. The revision resolves known exits and preserves an already computed index on eligible dynamic exits. The measurements establish the combined result. Estimating the contribution of each reduction would require separate controlled experiments.

Code growth also matters to the comparison. In the profile-guided build the executable text section grew from 701,879 bytes in plain SQLite to 1,118,759 bytes with the eight variants and stock interpreter. The larger executable still had the faster measured paths. These section sizes describe the whole application. We collected no instruction-cache or branch-miss counters.

This is a warm, in-memory, read-only experiment on one virtualized host. Training included the same queries and fixtures as measurement. The aggregate therefore describes reuse of this particular query mix. File-backed databases, different SQL or bindings, other C compilers, and other processors may give different results. Preparation, specialization and compilation are outside the timer, so the data supply no break-even reuse count. Bytecode identity, result equality and VM-step counts constrain the comparison but do not establish equivalence for every C program or SQLite configuration.

Reproducibility and authorship

The accompanying source archive contains the Rust evaluator and parser, C harness and SQL fixtures, snapshots, compiler and training commands, raw timed batches, baseline sources and table reconstruction scripts. Its README identifies the upstream SQLite release by cryptographic digest. A reader can check the published statistics offline before rebuilding either interpreter. The archive includes the editable manuscript and figure-generation code.

The author byline is Anonymous. Codex assisted with implementation, experiment execution, analysis and drafting. The manuscript is prepared for author and editorial review.

References

1. SQLite. *The SQLite bytecode engine*. Opcode reference.
2. C. Consel, J. L. Lawall and A.-F. Le Meur. A tour of Tempo: a program specializer for the C language. *Science of Computer Programming* 52, 341–370 (2004). Publisher record.
3. L. Hornof and J. Noyé. Accurate binding-time analysis for imperative languages: flow, context, and return sensitivity. *Theoretical Computer Science* 248, 3–27 (2000). Publisher record.
4. M. A. Ertl and D. Gregg. Optimizing indirect branch prediction accuracy in virtual machine interpreters. *PLDI*, 278–288 (2003). Publisher record.
5. C. F. Bolz, D. Kurilova and L. Tratt. Making an embedded DBMS JIT-friendly. *ECOOP*, 4:1–4:24 (2016). Publisher record.
6. SQLite. *Status parameters for prepared statements*. C interface documentation.
7. Free Software Foundation. *GCC 13.3: Options that control optimization*. Compiler manual.
8. Free Software Foundation. *GCC 13.3: Object size checking*. Compiler manual.