Specializing a Rewrite Evaluator for SQLite Bytecode

Anonymous

Abstract We implemented an evaluator that specializes explicit rewrite rules for SQLite bytecode. Programs from SQLite 3.50.4 remain fixed while table rows and parameter bindings vary. On a filtered scan, linking the residual rule actions reduced mean execution time from 32.60 to 18.99 microseconds compared with an evaluator that had already decoded each instruction. The paired time ratio was 1.72× (95% interval 1.68–1.75). Unmodified SQLite supplied the bytecode and independently reproduced all timed result checksums. A separate evaluator pass reused field values across residual actions and reduced scan time by a further 11% in a matched experiment.

1. The question left by the comparator

In an earlier study, specializing the second integer comparison in an index seek left the whole-query improvement unresolved. The primary stock-to-specialized time ratio was 1.0244, with a 95% paired interval of 0.9875–1.0755. That experiment changed one C comparison path. It did not vary interpreter dispatch and cannot identify the bytecode architecture as the cause of the result.

SQLite translates a SQL statement into a VDBE program. For a given prepared program, instruction addresses, opcodes, register indices and ordinary jump targets can be available before execution. Parameters, values read from a row and predicates on those values can vary on every invocation. These two sets of inputs make a prepared program a direct test of the first Futamura projection. The present experiment implements the projection for an explicit fragment of the VDBE. SQLite 3.50.4 supplies the programs and reference outputs. The Haskell program is a new evaluator for a rule language, independent of the earlier Tempo C front end. [1–3]

2. The unit of specialization

Write $E(R, P, s)$ for the evaluation of a fixed instruction sequence P from dynamic state s using the semantic rewrite rules R. The desired residual is $F_{R,P}(s) = E(R, P, s)$. Equality concerns the ordered observable result and termination behavior on the modeled domain. Specialization fixes the instruction at each address. It leaves dynamic comparisons and state changes to execution.

For example, consider the parameterized query

```
SELECT a+b, a*b+?2 FROM points
WHERE a>=?1
```

The exported program has 19 instructions. At address 4 it compares two registers. A less-than result or a NULL under this instruction's flags jumps to 13. Otherwise control falls through to 5. Specialization can bind the registers and both successors. The comparison still has to run for each row. At 13, advancing a cursor with another row returns to 3. One residual entry at 3 handles every iteration.

For a row with columns $(3, 4)$ and parameters $(2, 5)$ the residual emits $(7, 17)$. Changing the first parameter to 4 skips the same row. An empty table exits at the cursor's rewind instruction. All three executions use the same residual program.

3. Three ways to evaluate the rules

The generic evaluator selects a rule at every instruction. Predecoding selects a rule and binds its operands once per address, then executes the resulting actions through a program counter. The linked evaluator also resolves each static successor to its residual continuation. Dynamic conditionals select between continuations; back edges reference existing entries. A per-address bound on residual construction prevents loop unrolling as a function of table size.

This construction has a short local correctness argument. For each admitted address i, static rule selection produces the same action that generic matching would select for $P(i)$. Running its dynamic part on state s therefore produces the same observable event, next state and successor address. Relating control at i to the residual continuation for i gives a one-step simulation in both directions. Induction proves agreement on every finite prefix. The argument concerns the shared rewrite semantics. Correspondence of those rules with SQLite is a separate empirical question. The archive states the rule restrictions and proof assumptions.

The baseline matters. Comparing only against repeated rule matching charges the generic interpreter for work a straightforward interpreter can hoist when a statement is prepared. The predecoded evaluator still interprets the right-hand-side action at each visit. The linked evaluator shares its lowering into an executable closure and resolves its successors. Their timing contrast includes both effects. Measurements of previous interpreter specializers also distinguish reduction of interpreter overhead from total application time. [4–5]

4. Validation and measurements

Five query templates cover a filtered table scan, conditional arithmetic, a bound-value projection and two forms of scalar arithmetic. A pinned SQLite build produced the instruction sequences and ordered results. Across the main suite, all three modes agreed with SQLite on 565 executions and 5,126 rows per mode. An independent corpus used 11 table fixtures. All three modes matched its 461 executions, 1,057 rows and 193 NULL result cells. The release also rejected eight deliberately out-of-domain cases, including

integer overflow, unsupported comparison flags and external inputs outside the signed 64-bit range.

For timing, 24 paired blocks used all six orders of the three modes four times on the same x86-64 logical CPU. The scan had 96 rows and different threshold and arithmetic parameters. Each timed process performed 256 scans or 49,152 scalar executions. Each process warmed its program and inputs, consumed every returned value and included row and query boundaries in the checksum. GHC 9.6.6 compiled each mode into the same executable with optimization enabled. The primary statistic was the geometric mean of the blockwise ratio of predecoded to linked time; intervals resampled matched blocks 10,000 times. Construction cost and native SQLite time are outside this endpoint.

Program	Ratio	95% interval
Filtered scan	1.72	1.68–1.75
Scalar CASE	2.04	2.00–2.08
Projection	2.34	2.21–2.48

Table 1: Release predecoded time divided by linked time. Ratios above one favor linking.

The original scan measured 1.80× (1.74–1.87). Independent validation exposed a parser bug for textual integers outside the signed 64-bit range. After correcting input admission, we repeated the entire protocol. On the release, the predecoded and linked strategies averaged 32.60 and 18.99 microseconds per scan. Relative to the generic textual matcher the release ratio was 64.96× (63.72–66.19). That comparison includes repeated textual matching and operand decoding; it describes the cost of this generic interpreter. Native SQLite independently reproduced the checksum and row count of all 432 timed processes across both campaigns.

The effect applies to a warm, restricted evaluator over an immutable table snapshot. It does not measure B-tree page access, locking, sorting, writes, concurrent schema changes or a full SQLite connection. The cost of specialization per fresh statement and the execution count needed to recover it remain unmeasured.

5. Sharing reads in the residual semantics

The field-sharing pass runs on the residual register and cursor actions generated from the rules. It leaves the SQLite instruction sequence intact. At each program point a dataflow fact identifies a register containing a particular field of the current row. Assigning to a register kills its old fact. Moving or opening a cursor kills the cursor facts. At a join, only facts present on every incoming path survive. A read in this immutable snapshot model has no observable side effect.

For the 19-instruction scan, the pass eliminates the reads of column `a` at addresses 5 and 8: register 1 already holds the same value. At address 9, column `b` is available in register 5, so the action copies it to register 6. Over the archived batch of 92 queries, the scan visits 8,832 rows and selects 4,695. The original graph therefore performs $8832 + 4 \cdot 4695 = 27612$ field reads. The transformed graph performs $8832 + 4695 = 13527$. This count follows from the executed paths and the two action graphs.

In a separate paired experiment, both arms used the same executable and fully linked evaluator. Sharing reduced mean scan time from 18.11 to 16.10 microseconds per query. The paired baseline-to-sharing ratio was **1.125** (95% interval 1.106–1.141). The CASE control had no eligible reads and gave 0.983 (0.927–1.041). Each workload had 24 balanced paired blocks. This comparison isolates the residual dataflow pass within that executable. The main and supplementary ratios come from distinct experiments and should be interpreted separately.

6. Extending the rule evaluator

A small test demonstrates how the rule language can bind additional static information. Prepending a rule for parameter 1 with value 17 removes its dynamic read from the scan. Parameter 2 remains dynamic. All three strategies agreed with SQLite when parameter 1 was bound to 17 over 92 executions and 3,220 rows. If such a specialization is reused after rebinding, equality of the bound value must be part of the admission check.

A further extension is to carry static facts about registers across residual labels. A join retains only facts common to all incoming paths. A known value can simplify the right-hand side of a semantic rule; an unknown SQL value retains its NULL, integer, real or other applicable cases. Specializing only a finite number of abstract contexts per original address bounds residual growth. This extension needs tests that place different values on both sides of each join.

Finally, a production path must associate residuals with the exact prepared-program version and its semantic assumptions. SQLite can reprepare statements after a schema change or certain binding changes. A residual must be rebuilt or invalidated at that boundary. A guard on an unsupported runtime value can transfer execution to the general evaluator only if it reconstructs the correct live state and preserves each preceding effect. The present prototype admits integer and NULL values, read-only snapshots, and the operand patterns specified in the rule file. [2]

Contributors. A human requester specified the research question. OpenAI Codex implemented the evaluator, ran the experiments and drafted the report.

References

1. SQLite. The SQLite Bytecode Engine. See also release 3.50.4.
2. SQLite. Compiling an SQL Statement.
3. Y. Futamura. Partial Evaluation of Computation Process, An Approach to a Compiler-Compiler. Reprint, 2004.
4. C. Fallin and M. Bernstein. Partial Evaluation, Whole-Program Compilation. PLDI, 2025.
5. C. F. Bolz, D. Kurilova and L. Tratt. Making an Embedded DBMS JIT-friendly. ECOOP, 2016.