# Specializing SQLite Index Comparisons and a Rule-Based Bytecode Evaluator

Anonymous

**Abstract** Specializing the second integer field of a SQLite 3.50.4 index comparison gave a stock-to-specialized time ratio of 1.0244 on a warm four-row query (95% paired interval 0.9875–1.0755). We then specialized a Haskell evaluator whose rewrite rules describe SQLite bytecode. With a fixed program and dynamic rows and parameters, linked evaluation was 1.72× faster than predecoded evaluation on a filtered scan (1.68–1.75). In a separate experiment, sharing field values across residual actions reduced scan time by 11.1% (ratio 1.125, interval 1.106–1.141). Checksums and row counts agreed with native SQLite in all 528 timed evaluator sessions. The measurements establish a gain within the evaluator and leave the native whole-query advantage unresolved.

## 1. Two specialization questions

Consider an index on tenant and timestamp. SQLite already compares integer first fields directly during an index seek. When two tenants agree and the search key has another field, the implementation calls the general comparator with the first field skipped. We first ask whether specializing this continuation changes the execution time of a complete prepared query in the native engine [1].

The native experiment holds the interpreter fixed, so a small timing difference cannot identify bytecode dispatch as its cause. Our second experiment asks which work a Haskell evaluator can remove once the bytecode program is fixed. We time that evaluator with dynamic inputs, using unmodified SQLite to generate programs and check results.

## 2. A contract for an equal prefix

Write $(a, b)$ for the first two stored fields and $(x, y)$ for a two-field search key. Additional stored fields are allowed. On the selected path, $a = x$, both second fields represent integers and the second sort direction $d$ is ordinary ascending or descending. Define $r$ as the sign of the mathematical comparison of $b$ with $y$. The C code compares signed 64-bit operands without subtraction. SQLite encodes integers at several widths, including zero-byte payloads for zero and one [2].

For $r \neq 0$ the required return is $r$ in ascending order and $-r$ in descending order. The prior equality flag is preserved. For $r = 0$, SQLite sets the unpacked search record's `eqSeen` flag and returns its `default_rc`. The caller can use different defaults to distinguish positions within an equal prefix. An unconditional zero on equality would change observable behavior. All three cases preserve the prior error field on valid inputs to the integer path.

For dynamic state $u$ and direction $d$, the generated function must satisfy $C_d(u) = C(d, u)$ on the return and updated state. For $d$ fixed, the guard must select a subset of the equalizer of the stock and residual observation maps. The copied decoder and comparison establish the strict-order cases. The copied equality exit establishes the equal-prefix case. A rejected guard passes the original inputs to stock. This argument assumes a well-formed record satisfying SQLite's entry conditions.

## 3. From the amalgamation to the residual C

We first specialized SQLite's general comparator directly with the Haskell reconstruction of Tempo [3], fixing its skip argument to one. The tool parsed 2,574 functions, encountered shadowed object names and emitted a forwarding wrapper. The *plain* build redirects the integer-first tie site to that wrapper, which GCC can inline.

For the working candidate we wrote an adapter around the integer comparison and equality exit extracted from the digest-verified amalgamation. Dispatch requires two search fields, an integer-valued second operand and ordinary second-field order. After a tie, the adapter checks the packed header, serial tag and payload extent, uses SQLite's integer decoder, and falls back to the stock tail when checks fail. The storage class is dynamic even for columns with numeric affinity [4].

Two fresh Haskell processes parse the augmented translation unit (2,578 functions) and bind the sort direction to ascending or descending. The builder checks that the residuals eliminate the direction argument and branch, restores internal linkage and compiles them into SQLite. Selecting the tie path and writing its guard required manual work.

The *dynamic* control passes a direction read from the search metadata to the guarded C model. The *literal* control passes zero or one at the same site. GCC 14.2 inlined the Tempo residuals into two 426-byte comparator functions. The dynamic build produced a 459-byte function and a five-byte jump. The literal build produced 279- and 282-byte functions sharing a 193-byte helper. These are static code sizes.

## 4. Measuring native prepared queries

The primary query reads four timestamps and payloads for one tenant from a half-open ordered range. A 131,072-row table has 128 timestamps per tenant. Its covering index declares `(tenant,tick,payload)` and uses two constrained search fields. A second experiment declares

exactly `(tenant,tick)`, checks the schema, and projects `tick,rowid`. Each process verifies the covering-index plan [5,6].

GCC 14.2.0 built all libraries with `-O2 -DNDEBUG -fPIC`. Each fresh process ran on logical CPU 8 of an AMD EPYC 9V74, opened the same read-only file for its workload, disabled memory mapping, set a 32 MiB page cache and warmed the B-tree. Before timing it checked 512 parameter sets row by row. The clock covered binding, stepping, reading both values into an ordered checksum and resetting. The full count and checksum had to match an independent answer.

Eight workloads ran in 24 matched blocks, using each permutation of the four builds once per workload. Every build in a block received the same parameters. Each session had 32,768 executions (8,192 for 64-row ranges). The 768 main sessions plus two separate 24-block paired experiments gave 864 timed processes. Separate instrumented runs counted ties, hits and fallbacks.

For paired times $S_i, C_i$, the prespecified ratio is $\exp(\mathrm{mean}_i \log(S_i/C_i))$. Values above one favor the denominator. Two-sided 95% percentile intervals used 10,000 whole-block bootstrap samples and a fixed seed. Arithmetic paired savings estimate amortization. Secondary intervals describe this build and host without multiplicity correction.

## 5. Native comparator results

A C harness compared 149,784 valid inputs with stock and an independent order calculation. The exact return, sign, equality flag, default return and error code all agreed. There were 39,624 equal prefixes, 95,544 residual hits and 1,056 fallbacks inside the selected tail. Cases included width boundaries, both directions, multibyte headers and both initial equality states. Damaged records and allocation failures were outside the corpus.

The public C API produced identical results for 5,019 observations per build, including 24,122 ordered rows, connection and error state, and matching final file hashes. Cases covered mixed types, collations, three-field keys, schema reprepare and transactions. Of 54,424 ties in the counted candidate, 19,421 used the new path and 35,003 the generic continuation. All 448 three-field ties used the latter.

**Primary result.** Stock averaged 3,278.3 ns/query and the candidate 3,190.5. The paired geometric ratio was 1.0244 (0.9875–1.0755). Each session returned 131,072 rows. A separate counted session handled all 249,299 first-field ties on the fast path, or 7.61 per query.

**Exactly two index columns.** The change in index and projection gave a stock-to-candidate ratio of 0.9882 (0.9725–1.0019). The candidate handled all 248,494 ties in the instrumented session. The point estimate favored stock. Because the stored entry width and the projected value both changed, the difference between the two experiments cannot be assigned to either factor separately.

**Compiler controls.** Primary ratios were 1.0055 (0.9871–1.0235) for dynamic/candidate and 1.0209 (0.9905–1.0660) for stock/plain. The separately paired literal/candidate ratio was 1.0152 (0.9736–1.0651). The measurements leave the incremental advantage of Tempo's generated code unresolved.

With 2,048 rows per tenant the candidate made 11.58 fast comparisons/query and gave a point ratio of 0.9893. The rowid lookup bypassed the comparator and gave 0.9924 (0.9554–1.0275). Mixed integer/text keys took the fast path on 43.6% of ties, yet favored stock at 0.9333 (0.8768–0.9808). The one-row-per-tenant workload also favored stock. Hit counts alone do not predict a query-time gain.

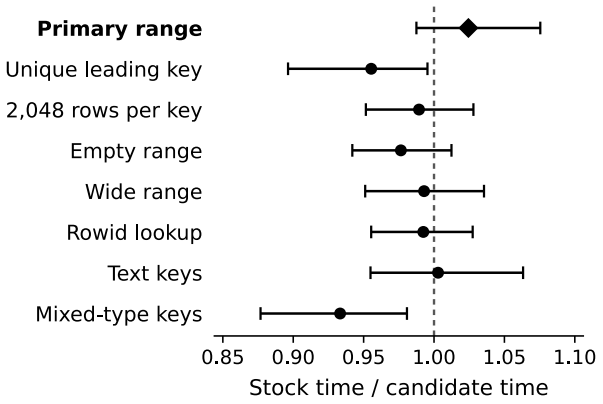



Figure 1: Stock-to-candidate elapsed-time ratios for complete queries. Points are paired geometric means. Bars show 95% paired bootstrap intervals over 24 blocks per workload.

**Mean query times and path counts.** Times are ns per complete query, averaged over the 24 timing blocks. Hits per query come from separate counter builds.

| Workload | Stock | Candidate | Hits/query |
|---|---|---|---|
| Primary | 3,278 | 3,191 | 7.61 |
| Unique tenant | 2,902 | 3,067 | 1.00 |
| 2,048 per tenant | 3,311 | 3,353 | 11.58 |
| Empty range | 2,810 | 2,884 | 7.54 |
| 64-row range | 8,316 | 8,368 | 7.54 |
| Rowid | 2,766 | 2,793 | 0.00 |
| Text keys | 3,573 | 3,551 | 0.00 |
| Mixed keys | 3,276 | 3,548 | 3.32 |

**Generation cost.** Preprocessing and the two Haskell specializations took 0.218, 1.808 and 1.369 seconds (3.395 total). C compilation and linking took 20.94 seconds for stock and 20.84 for the candidate in a four-job parallel build. Concurrent times cannot isolate incremental compile cost.

The primary arithmetic saving was 87.8 ns/query (−42.1 to 265.7). Its point estimate would recover generation time

alone after 38.7 million queries. The interval admits a loss, so no finite upper bound on recovery follows. Prebuilt users incur no local generation cost.

## 6. Partial evaluation of the rewrite rules

SQLite's Virtual Database Engine (VDBE) executes the program produced when a statement is prepared [8]. We implemented a separate Haskell evaluator for a read-only fragment, with instruction semantics in an external rule file. Let $E(R, P, s)$ denote evaluation using ordered rules $R$, program $P$ and dynamic state $s$. Partial evaluation at fixed $(R, P)$ produces a residual $F_{R,P}$ satisfying $F_{R,P}(s) = E(R, P, s)$ on ordered output for admitted terminating executions. This is the first Futamura projection applied to the modeled machine [9]. This distinction also arises in work on interpreter compilation and embedded databases [10,11].

Each rule matches an opcode and five operand fields. Its right-hand side specifies register writes, arithmetic, tests, cursor movements, result emission and successors. Operand locations and static successors are known. The values they identify remain dynamic. Two rules from the executable specification illustrate the distinction. A dollar sign binds an operand, and `fall` advances to the next address. The addition rule is split across lines here for readability.

```
Goto 0 $target 0 - 0 => (go $target)
Add $r1 $r2 $dst - 0 =>
  (set $dst (+ (reg $r2) (reg $r1)) fall)
```

In the 19-instruction scan below the register addresses can be bound once, while each fetched row supplies different operands.

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

At address 4 a comparison reads the current row's value of `a` and the bound lower limit. Its fixed target is address 13, taken on a smaller value or on NULL under the recorded flags. Otherwise execution falls through to 5. Address 13 advances the cursor and returns to 3 if another row exists. For row $(3, 4)$ and bindings $(2, 5)$ the program emits $(7, 17)$. Changing only the lower limit to 4 skips that row. Both executions use the same residual graph.

The **generic** evaluator matches and instantiates a rule at every instruction. The **predecoded** evaluator matches once per address, then interprets the stored action at each visit using a numeric program counter. The **linked** evaluator lowers each action to a shared executable closure whose static successors reference their residual entries. A backward edge refers to an existing entry, so program construction is finite even when the number of rows is unknown. The predecoded-to-linked contrast includes both shared action lowering and resolved successors.

To see why linking preserves ordered output, relate control at each address $i$ to its residual entry with the same dynamic state. Ordered static matching selects the same action in both executions. The two evaluations of that action perform the same state update, emit the same row if any, and select related successors; a silent jump can be collapsed when linking. Induction gives equal output on finite terminating executions. Agreement of the rule model with native SQLite is an additional empirical obligation. A priority rule mapping parameter 1 to the literal 17 left parameter 2 dynamic. All three evaluators matched SQLite over 92 executions and 3,220 rows without changing the executable.

## 7. Evaluator timings and validation

GHC 9.6.6 compiled all three strategies into one optimized executable. Each fresh process ran one untimed batch, then evaluated dynamic bindings against the same immutable table snapshot and consumed every returned field into an ordered checksum. The primary scan contained 96 rows. Each process ran 256 scans or 49,152 scalar executions. Each workload used 24 matched blocks, with each of the six orders occurring four times. The interval calculation used the same paired geometric ratio and 10,000 whole-block resamples as the native study. These are warm execution times; setup and specialization costs are outside the clock.

**Released evaluator.** Times below are arithmetic means in microseconds per execution. Ratios and 95% intervals compare paired predecoded and linked times.

| Program | Predecoded | Linked | Ratio (95% interval) |
|---|---|---|---|
| Filtered scan | 32.598 | 18.990 | 1.72 (1.68–1.75) |
| Scalar CASE | 0.257 | 0.126 | 2.04 (2.00–2.08) |
| Projection | 0.136 | 0.058 | 2.34 (2.21–2.48) |

The first scan campaign gave 1.80 (1.74–1.87). Independent tests found an external input parser that wrapped out-of-range decimal integers. After fixing admission we repeated the entire schedule, obtaining the release results above. Against the generic textual matcher the release scan ratio was 64.96 (63.72–66.19). That larger ratio includes repeated pattern matching and operand decoding. The predecoded baseline already eliminates this work.

Across five query templates the three evaluators matched SQLite on 565 executions and 5,126 ordered rows per mode. A second suite used 11 table fixtures, including duplicates, empty tables and signed boundaries. Each mode matched 461 executions, 1,057 rows and 193 NULL cells. The release rejected eight out-of-domain cases per mode, including arithmetic overflow that SQLite promotes to REAL. Native SQLite independently reproduced checksums and row counts for all 432 timing sessions of the original and release campaigns.

## 8. Reusing field values in residual actions

A second pass operates on the linked evaluator's action graph. A fact $(r, c)$ at a node means that register $r$ holds field $c$ of the current row on every incoming path. A load

or copy can establish a fact. A register write invalidates its previous fact, a cursor movement invalidates facts about the old row, and a join takes the intersection of incoming facts. Replacing a redundant read therefore preserves the value on each admitted execution. This relies on immutable, effect-free field reads in the model.

In the scan, address 3 reads column `a` into register 1. The pass eliminates repeated reads into that same register at 5 and 8. At 9 it replaces a read of `b` with a copy from register 5. In the archived 92-query batch, 8,832 row visits and 4,695 selected rows imply $8,832 + 4(4,695) = 27,612$ modeled reads before and $8,832 + 4,695 = 13,527$ after, a derived reduction of 51.0%.

In a separate 24-block experiment both arms ran in the same executable and used the linked evaluator. Sharing reduced mean scan time from **18.11 to 16.10 µs**, a paired baseline-to-sharing ratio of **1.125** (95% interval **1.106–1.141**), or 11.1% less elapsed time. The CASE control had no eligible reads and gave 0.983 (0.927–1.041). All 96 sessions matched the SQLite checksums and row counts, bringing the evaluator total to 528. The linking and sharing ratios use separate builds and matched comparisons; the data supply no combined ratio against stock SQLite.

## 9. Interpretation and replication

The native comparator handled every eligible tie in the primary counter runs, yet both native timing intervals span one. The Haskell results locate recurring work in the evaluator. Linking avoids reinterpreting fixed actions, and sharing avoids fetching an available field value. The unchanged CASE control helps distinguish the sharing effect. The timing evidence covers one host, one read-only snapshot model and its INTEGER/NULL fragment. Native SQLite remained the reference for bytecode and outputs throughout the evaluator study.

For native C specialization, scope-aware renaming of shadowed locals would allow a direct test on the original tail. Replications with varied code layouts and a second processor would test the small native timing differences [7]. For the rule evaluator, the next step is to model storage, effects and errors. A production residual also needs invalidation when SQLite reprepares a statement after a schema change or a plan-sensitive binding change [12]. A guard transferring control to general execution must reconstruct live register and cursor state and preserve any effects already committed.

The replication package includes both experiments, their semantic and source code, the pinned SQLite release, exact programs and measured evidence. Reproduction instructions cover native comparator checks, all three evaluator modes, the field-sharing pass and the paired analyses.

**Contributor disclosure.** Anonymous manuscript. A human requester set the research objective. OpenAI Codex developed the integration and tests, ran the experiments and drafted the manuscript.

## References


1. SQLite. *SQLite 3.50.4 release log and SHA3-256 digest of the amalgamation*. Release record
2. SQLite. *Database File Format*, §2.1. Record format
3. Inria / IRISA. *Tempo Specializer*. Project documentation
4. SQLite. *Datatypes in SQLite*, §§2–3. Storage classes and affinity
5. SQLite. *Query Planning*, §1.6–1.7. Multicolumn and covering indexes
6. SQLite. *EXPLAIN QUERY PLAN*. Plan diagnostics
7. C. Curtsinger and E. D. Berger. “STABILIZER: Statistically Sound Performance Evaluation.” ASPLOS, 2013. Author manuscript
8. SQLite. *The SQLite Bytecode Engine*. Instructions, registers and cursors
9. Y. Futamura. *Partial Evaluation of Computation Process, An Approach to a Compiler-Compiler*. 2004 reprint. Article
10. C. Fallin and M. Bernstein. *Partial Evaluation, Whole-Program Compilation*. PLDI, 2025. Author manuscript
11. C. F. Bolz, D. Kurilova and L. Tratt. *Making an Embedded DBMS JIT-friendly*. ECOOP, 2016. Article
12. SQLite. *Compiling an SQL Statement*. Statement compilation and reprepare