Specializing the Second Field of a SQLite Index Comparison

Anonymous

Abstract SQLite already compares integer first fields directly during an index seek. We used a Haskell partial evaluator to specialize the comparison after a tie. For a warm four-row range query on a covering index with a payload column, the stock-to-specialized time ratio was 1.0244 (95% paired interval 0.9875–1.0755). An index with exactly two declared columns gave 0.9882 (0.9725–1.0019). All 149,784 direct comparison tests agreed with stock. These measurements establish feasibility, while leaving a whole-query speed advantage unresolved.

1. What does a second-field specialization buy?

Consider an index on tenant and timestamp. To locate a range within one tenant, SQLite compares a key supplied by the query with keys stored on B-tree pages. SQLite 3.50.4 already has a fast path for integer first fields [1]. When two tenants differ, this comparison determines their order. When two tenants agree and the search key has another field, the implementation calls a general comparison routine with instructions to skip the first field. Can a specialized integer comparison at that point reduce the elapsed time of a complete prepared query?

This case makes it possible to test three separate propositions. The transformed comparator must preserve SQLite's return value and state. The generated path must actually run on the workload. Finally, removing work from the comparator must save enough time to affect binding, executing and consuming results. The first two propositions permit direct checks inside SQLite. The third requires whole-query timings against the optimized library and against carefully matched implementations.

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 fields can follow the stored prefix. On the selected path, $a = x$, both second fields represent integers, and the second sort direction d is either ascending or descending. Define r by the mathematical ordering of b and y. It is -1, 0 or 1. The C implementation uses signed 64-bit ordered comparisons, and never subtracts the operands. SQLite stores integers at several widths and encodes zero and one with serial types that use no payload bytes [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.

The specialization is an instance of partial application of a state transformer. Let $C(d, u)$ be the guarded comparison, where u comprises both arguments and mutable comparison state. For $d = 0, 1$, the generated function C_d must satisfy $C_d(u) = C(d, u)$, with equality on both the returned integer and the resulting state. In the category of sets, fixing d restricts C to the fiber over d of the projection from directions and inputs to directions. For valid calls in that fiber, the guard must select a subset of the equalizer of the stock and generated observation maps. This gives a separate obligation for the adapter. The copied decoder and comparisons establish the strict-order cases, the copied equality exit establishes the equal-prefix case, and a rejected guard calls the original routine with its original inputs. This argument applies to well-formed records satisfying SQLite's entry conditions. The implementation and finite tests below provide evidence about the concrete integration.

3. From the amalgamation to the residual C

We used the C source-to-source path of the Haskell reconstruction of Tempo, a model-directed C specializer [3]. We first attempted to fix the skip argument to one directly in SQLite's general comparison routine. The Haskell tool parsed 2,574 function definitions and reported shadowed object names in the target. It conservatively emitted a forwarding call. The resulting build, called *plain* below, redirects the integer-first tie site to the generated wrapper. GCC can inline that wrapper. This experiment exposes a concrete limitation in specializing the unmodified source.

For the working candidate, we supplied an adapter by hand. A build script verifies SQLite's published source digest, extracts its integer comparison and equality exit, and inserts both into the adapter. At the existing integer-first dispatch site, the added guard requires exactly two search fields, a second integer-valued operand and an ordinary second-field sort direction. After a first-field tie, the adapter checks the packed header, integer serial tag and payload extent. SQLite supports heterogeneous storage classes, so the declared column type alone would be insufficient [4]. Rejected cases retain the general tail. Both directions use SQLite's existing integer decoder.

Two fresh Haskell invocations parse the entire augmented translation unit, containing 2,578 function definitions. One fixes the direction to ascending and one to descending. The build checks that each residual eliminates the direction argument and branch, restores internal linkage, and places the residual behind its direction-specific first-field comparator. The generated C is compiled and linked into the measured SQLite shared library. The Haskell implementation performs

the residualization. Selecting the tie path and writing its guard required manual work.

Two additional controls distinguish source transformation from code generation by GCC. The *dynamic* build has the same dispatch and guard but passes a direction read from the search metadata. The *literal* build passes the constants zero and one to the same C function. On x86-64, GCC 14.2 inlined the Tempo residuals into two 426-byte comparator functions. In the dynamic build it emitted one 459-byte function plus a five-byte jump. In the literal build the 279- and 282-byte functions share a 193-byte helper. These object-code differences establish that GCC produced distinct implementations. Timing determines their cost.

4. Measuring complete prepared queries

The primary query returns the timestamp and payload for one tenant over a half-open timestamp range, in timestamp order. A file-backed table contains 131,072 rows with 128 successive timestamps per tenant. Each primary query returns four rows. The main index declares three columns, `(tenant,tick,payload)`. The two constrained fields form the search key, and the payload makes the query covering. A separate experiment uses exactly two declared index columns, `(tenant,tick)`, and projects `tick,rowid`. Each session in that experiment checks the declared columns. Both queries use a covering index [5]. Every process verifies its access plan using the pinned release's diagnostic interface [6].

All builds used GCC 14.2.0 with `-O2 -DNDEBUG -fPIC`. A Linux x86-64 process was pinned to logical CPU 8 of an AMD EPYC 9V74. Each fresh process loaded one library, opened the same read-only database for its workload, warmed the B-tree with a scan, and used a 32 MiB page-cache setting with memory mapping disabled. A session audited the first 512 parameter sets row by row before timing. During timing it bound parameters, stepped to completion, consumed both output values into an order-sensitive checksum, and reset the statement. Preparation, input generation and warm-up were outside the timed region. The final row count and checksum had to match a separately computed answer for the entire session.

Each of the eight workloads ran in 24 matched blocks. Within a block every build received the same parameters. All 24 permutations of the four main builds occurred once per workload. A session comprised 32,768 executions, or 8,192 for the 64-row range. The main experiment comprised 768 timed processes. Two separate 24-block experiments compared literal C with the candidate on the primary workload and compared stock with the candidate on the exactly-two-column index. Separate instrumented builds counted ties, fast comparisons and fallbacks outside the timing runs.

The primary statistic was fixed before comparative execution. For paired block times S_i and C_i, the reported ratio is $\exp(\mathrm{mean}_i \log(S_i/C_i))$. A ratio above one favors the candidate in a stock-to-candidate comparison. We resampled whole paired blocks 10,000 times with a fixed seed and report two-sided 95% percentile intervals. Arithmetic paired time savings give a separate estimate for amortization. Each interval describes repeated sessions on one host and build. Secondary intervals are descriptive and use no multiplicity correction.

5. Correctness and performance

An internal C harness compared the candidate to both SQLite and an independent order calculation on 149,784 valid inputs. It checked the exact return, order, equality flag, default return and error code, including 39,624 equal prefixes. The two residuals handled 95,544 cases. Another 1,056 entered a selected tail and fell back. The remainder used stock dispatch or finished at the first field. All comparisons agreed. Cases covered signed width boundaries, both directions, one- and multibyte headers, and pre-existing equality state. Damaged records and forced allocation failures are outside this result.

A separate public C API test compared 5,019 observations per build, including 24,122 ordered rows, status and error details, and connection state. Final database SHA-256 digests also agreed. The candidate and all three controls agreed with stock. Tests covered duplicate and empty ranges, reverse scans, collations, mixed storage classes, transactions, schema reprepare and three-field search keys. A counted run of the candidate recorded 19,421 fast comparisons and 35,003 generic continuations. All 448 ties in the three-field group used the generic continuation.

Primary result. Mean times were 3,278.3 ns per query for stock and 3,190.5 ns for the candidate. The paired geometric time ratio was 1.0244 (95% interval 0.9875–1.0755). The stock and candidate returned the same 131,072 result rows per primary session. An instrumented session recorded 249,299 fast comparisons in 32,768 queries, or 7.61 per query, with no generic continuations at those ties. The interval includes one despite frequent execution of the generated code.

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. On the primary workload the dynamic-to-candidate ratio was 1.0055 (0.9871–1.0235). In the separate literal comparison the literal-to-candidate ratio was 1.0152 (0.9736–1.0651). The stock-to-plain ratio was 1.0209 (0.9905–1.0660), even though the plain residual forwards to the stock implementation. These comparisons leave the incremental whole-query advantage of the Haskell-generated code unresolved.

The 2,048-rows-per-tenant workload had 11.58 fast comparisons per query and a point ratio of 0.9893. The high hit count alone did not predict a benefit. At the other extreme, the rowid lookup bypassed the comparator and gave 0.9924 (0.9554–1.0275). The mixed integer/text workload took the fast path on 43.6% of ties and gave 0.9333 (0.8768–0.9808), favoring stock. The unique-tenant workload also favored stock, with one fast comparison per query. These observations limit the usefulness of a dispatch rule based solely on integer types and hit counts.

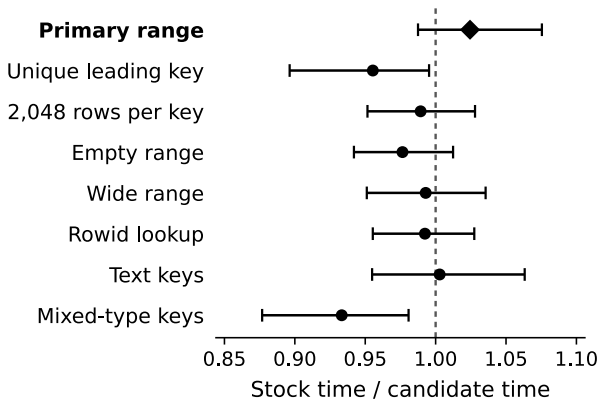

Figure 1: Stock-to-candidate elapsed-time ratios for complete queries. Points are paired geometric means. Bars show 95% paired bootstrap intervals over 24 blocks per workload.

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

6. Cost, scope and next experiment

The one-time preprocessing and two Haskell specializations took 0.218, 1.808 and 1.369 seconds, respectively. Their total was 3.395 seconds. Subsequent C compilation and linking took 20.94 seconds for stock and 20.84 for the candidate in a four-job parallel build. Those concurrent stage times do not isolate an incremental compilation cost. For an already distributed binary, the end user incurs no local generation time.

On the primary workload, the arithmetic mean saving was 87.8 ns per query (95% interval −42.1 to 265.7 ns). If the point saving persisted, recovering the generation time alone would require approximately 38.7 million queries. The interval includes a loss, so there is no finite upper confidence bound for recovery. This estimate excludes C compilation, integration effort and distribution costs.

The result supports source-level specialization inside a complete SQLite library. Its performance evidence is limited to warm read-only queries, one compiler configuration, one host and fixed binary layouts. The full upstream test suite, write-heavy workloads and cold-cache execution remain outside the evaluation.

The next experiment should address two independent uncertainties. First, implement scope-aware renaming of shadowed C locals, then repeat the direct specialization attempt on the unmodified general comparator. The acceptance test is a residual whose active path actually simplifies the original tail and passes the same state and SQL checks. Second, test each build across matched, independently varied code layouts and on a second processor before making a deployment decision. Prior performance experiments document the dependence of small timing differences on layout [7]. Keeping the same SQL contract, rowid control and paired analysis would distinguish a repeatable query gain from the observed build-to-build variation.

The replication archive contains the pinned SQLite amalgamation, Haskell sources and executable, generated residuals, control builds, disassembly summaries, correctness tests, timed binaries, raw block timings, counters and analysis code. The root README gives commands for rebuilding and checking the study offline.

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