FastCompose: Eliminating Compilation Cold Starts in Query Execution with Composition

Venkatraman Govindaraju
Amazon Web Services
East Palo Alto, USA
gvraman@amazon.com

Yankun Shen
Amazon Web Services
East Palo Alto, USA
syankun@amazon.com

Kiran Chinta
Amazon Web Services
East Palo Alto, USA
kkchinta@amazon.com

Shermal Fernando
Amazon Web Services
Berlin, Germany
shermalf@amazon.de

Pascal Pfeil
Amazon Web Services
Munich, Germany
pfeip@amazon.de

Orestis Polychroniou
Amazon Web Services
New York, USA
orestis@amazon.com

Naresh Chainani
Amazon Web Services
East Palo Alto, USA
nareshkc@amazon.com

Abstract

Compilation-based query execution produces optimized machine code per query but introduces a cold-start problem: when the compiled code is not cached, the query stalls during compilation, delaying data processing by up to orders of magnitude relative to the query's execution time. This overhead dominates short-running queries and creates latency variability for both interactive analytics and ETL pipelines. We introduce *composition*, a complementary technique in which query-time code generation emits only lightweight glue code to arrange pre-compiled operators into a query-specific execution plan, rather than re-emitting or compiling any operator logic, at a fraction of the cost of full compilation. Composition eliminates the cold-start stall while compilation catches up in the background and takes over for peak performance. We implement composition in Amazon Redshift through *FastCompose*, which enables both modes from a single code-base without relying on a separate fallback engine for cold runs. Compared to compilation-only cold starts (with caching and serverless compilation active), FastCompose achieves 7.0× speedup on TPC-DS 100 GB, 2.0× on TPC-DS 3 TB, 12.0× on TPC-H 100 GB, and 1.6× on TPC-H 3 TB. FastCompose is deployed across thousands of Redshift clusters. On production workloads, composition reduces cold-start dashboard load times by 3.5× and ETL duration by 1.9×, removing compilation as a bottleneck.

PVLDB Reference Format:
Venkatraman Govindaraju, Yankun
Shen, Kiran Chinta, Shermal Fernando, Pascal Pfeil, Orestis Polychroniou, and Naresh Chainani. FastCompose: Eliminating Compilation Cold Starts in Query Execution with Composition. PVLDB, 14(1): XXX-XXX, 2020.
doi:XX.XX/XXX.XX

1 Introduction

Modern analytical database systems rely on compilation-based query execution to achieve high performance [1, 2, 14, 15, 19]. The idea dates back to System R [7], and modern systems such as HyPer [23] and Umbra [4, 16, 24] compile query plans into native code, removing interpretation overhead and intermediate materialization. However, compilation can take several seconds, during which the query makes no progress on data processing. This overhead dominates short-running queries and creates latency variability for interactive analytics and ETL pipelines [10].

In Amazon Redshift [2, 14], the execution engine decomposes a query into *segments*, each covering a portion of the query plan up to a materialization or data redistribution boundary. Each segment is compiled independently into a shared library and cached. A query is *cold* when even a single segment misses the code cache, delaying the entire query. In steady state, Redshift's two-tier code cache (a local cache on each node and a remote cache backed by a serverless compilation service [8, 26]) serves the vast majority of queries from cached compiled code. However, binary upgrades invalidate local caches, and previously unseen segments (from new query patterns or adaptive re-optimization) miss even the remote cache. An analysis of the Redshift fleet [32] shows that over 87% of queries complete in under one second, making compilation overhead the dominant cost whenever a cold segment is encountered.

Existing mitigations such as compilation caching, constant parameterization, serverless compilation [8], and separate fallback engines [16, 24] reduce cold-start latency but do not eliminate it (see Table 3 in §2). The root cause is that all of these approaches still generate operator *logic* at query time. Whether the system emits C++ source, LLVM IR, or JVM bytecode, the generated code contains operator implementations that must be compiled.

We introduce *composition*, a complementary technique that generates only the *arrangement* of pre-existing logic rather than the logic itself. At query time, lightweight glue code (function calls, stack allocations, and control flow) wires pre-compiled operator implementations together and is compiled via LLVM into a shared library in milliseconds rather than the seconds required for full compilation. Composition removes compilation from the critical path: it provides immediate execution while compilation proceeds in the background, and the compiled object is cached for subsequent executions. Scalar expressions are handled identically: pre-compiled into the binary and called, never regenerated (§5).

We implement composition in Amazon Redshift [2, 14] through **FastCompose**. The mechanism works as follows: in Redshift, each

Proceedings of the VLDB Endowment, Vol. 14, No. 1 ISSN 2150-8097.
doi:XX.XX/XXX.XX

operator has a code generation method that emits C++ text describing the operator's logic. At build time, a preprocessor augments each method to also emit lightweight embedded domain-specific language (DSL) calls alongside the original C++ text. At query time, both outputs are produced in a single pass: the C++ text is submitted for serverless compilation, while the DSL calls generate LLVM IR glue code that is compiled into a shared library and executed immediately. This is fast because the glue code calls pre-compiled operator implementations through opaque pointers, hiding their internals from the compiler and eliminating the need to parse headers, instantiate templates, or optimize operator bodies.

Three build-time mechanisms enable this from a single operator codebase: an *Injector* that transforms existing code generation logic to support both modes, a *Composer* DSL through which the injected code produces lightweight glue code at query time, and a *Shielder* that hides operator internals behind stable interfaces. Together they avoid the need for duplicate engines and ensure semantic equivalence (§4). At runtime, FastCompose extends Redshift's two-tier code cache with a three-tier progression: immediate composed execution on a cache miss, cached composed libraries until the compiled object is available, and fully optimized compiled code for all subsequent executions. Because composition covers all production query execution paths, no query waits for g++ compilation.

This paper makes the following contributions:

- **Composition as a technique (§2–3).** We identify and formalize composition as a distinct execution technique that complements compilation, characterizing the key distinction (generating arrangement vs. generating logic) and the properties required to implement it in a compilation-based engine. This reframes the cold-start problem as solvable without a separate engine or custom compiler backend.
- **Single codebase realization (§4–5).** We present three mechanisms (Injector, Composer, Shielder) that enable composition and compilation from the same operator source code, avoiding the need for duplicate engines. This ensures semantic equivalence (§4) and halves the maintenance burden compared to multi-engine approaches.
- **Large-scale evaluation (§6).** On standard benchmarks, FastCompose achieves 7.0× speedup on TPC-DS 100 GB and 2.0× on TPC-DS 3 TB over a baseline that already includes compilation caching and serverless compilation. In production across thousands of clusters, composition reduces median per-segment compilation time by 61×. On a representative enterprise dashboard workload, cold-start latency improves by 3.5×, closing the gap between cold and warm query performance.
- **Production deployment and experience (§7).** FastCompose is deployed across thousands of Amazon Redshift clusters. The directive-based approach supports incremental adoption (operators are annotated one at a time with no runtime changes) and shifts most code generation defect detection from query runtime to build time. We report lessons from development and deployment, including the failure of a prior dual-engine approach and the engineering trade-offs that shaped the design.

Table 1: Terminology used throughout this paper.

Term	Definition
Build time	Building the Redshift binary (once per release).
Query time	When a query is received, planned, and executed.
Code generation	Producing C++ text and DSL calls (per segment).
Compilation time	When g++ compiles generated C++ into a `.so`.
Composition time	When LLVM generates `.so` from glue code IR.
Execution time	Time a segment's `.so` runs on compute nodes.
Cold query	One or more segments missing from the code cache.
Warm query	All segments served from the code cache.

2 Background

This section surveys query execution strategies, examines why compilation is expensive, and shows why existing mitigations fall short. Table 1 defines the key terms used throughout this paper.

2.1 Query Execution Strategies

Table 2 compares query execution strategies along per-tuple overhead, materialization cost, and startup latency.

Interpretation. The Volcano iterator model [11] processes tuples one at a time through virtual function calls at each operator boundary, incurring $O(N \cdot k)$ dispatches for k operators and N tuples.

Vectorization. MonetDB/X100 [6] amortizes this overhead by processing vectors of tuples per call, enabling SIMD and prefetching, at the cost of intermediate materialization between operators.

Compilation. Compilation fuses operators into a single loop, removing both interpretation overhead and materialization. The approach has a long history: Gamma [9] compiled predicates into machine language, HIQUE [19] generalized this to full operator code generation, and HyPer [23] advanced this strategy with LLVM-based compilation. Umbra [4, 16] further pushes this line with a custom IR and direct-to-x86 backend aimed at reducing compile time, and domain-specific IR designs [10, 13] explore similar trade-offs. However, compilation comes at the cost of cold-start latency (seconds for complex queries) and unbounded optimization time.

Composition. Composition, introduced in this paper, generates only the *arrangement* of pre-compiled operators through lightweight glue code, taking milliseconds to generate. Because operator calls are not inlined across boundaries, composed code does not match the peak performance of fully compiled code. Composition is therefore designed as a complement to compilation: it provides immediate execution for cold queries while compilation proceeds in the background, and the compiled code takes over once ready. The remainder of this section and section 3 develop this idea in detail.

2.2 Why Compilation Is Expensive

Despite its performance benefits, compilation introduces a challenge: cold queries can wait several seconds for compilation, during which no data is processed. The costs fall into two categories.

Frontend costs (always present). Systems that generate C or C++ source must invoke a compiler that parses the generated code, includes headers, and instantiates templates, even at the lowest optimization level (`-O0`). In C++ codebases, header inclusion alone can expand a few hundred lines of generated code into tens of thousands of lines. Template instantiation further multiplies this cost, as each

Table 2: Comparison of query execution strategies.

	Interpretation [11]	**Vectorization** [6]	**Compilation** [23]	**Composition (ours)**
Per-tuple overhead	High	Amortized (k/V)	None	Call overhead
Materialization	None	$O(k \cdot V)$/batch	None	Stack only
SIMD exploitation	Difficult	Natural	Auto-vectorization	Inherits from operators
Memory parallelism	None	High (batched prefetch)	Limited	Inherits from operators
Startup latency	None	None	High (unbounded)	Minimal
Parsing/template cost	None	None	Per-query (headers, templates)	None
Optimization scope	Per-operator	Per-operator	Unbounded (full pipeline)	Per-operator (at build time)
Implementation complexity	Low	Medium	High (codegen)	Low (reuses codegen)
Codebase	Single	Single	Codegen infrastructure	Single (shared with compilation)

type-specialized operator produces a full copy that the compiler must process. Systems that generate lower-level IR (e.g., LLVM IR) avoid these frontend costs but are harder to build, debug, and maintain [16].

One might avoid frontend costs by hiding operator internals behind opaque pointers (the pImpl pattern in C++), so the compiler never sees operator definitions. But this defeats the purpose of compilation: the compiler can no longer inline operators or optimize across their boundaries, giving up the performance advantage that motivated compilation in the first place.

Backend costs (optimization). When operators are fused into a single compilation unit, the compiler sees the entire pipeline as one large function body. Optimization passes such as inlining, loop unrolling, constant propagation, and register allocation operate on this fused code with superlinear cost in code size. Compilation-based engines rely on aggressive inlining (`always_inline`) to eliminate operator boundaries, and g++ follows these directives even at `-O0`. Simply lowering optimization levels does not help: the inlined code still expands, and tuning compiler flags is fragile. A compiler upgrade can change inlining heuristics and invalidate carefully tuned settings.

A concrete walkthrough. Consider TPC-DS Q64, whose largest segment implements a deep hash-join chain with resizeable hash tables and prefetching. The code generator emits 1.6K lines of operator and join-condition C++ (annotated `always_inline`). The preprocessor expands headers to produce a 362K-line translation unit. g++ then instantiates templates within this unit into type-specialized code and optimizes the resulting fused function. Optimization on the resulting fused function grows superlinearly in code size, producing multi-second compilation for sub-second runtime (Table 6). Wide projections, deep join chains, and multi-distinct aggregations amplify this, explaining the results in Table 5. Per-segment cost also compounds across the many segments per query (Q64 alone has 53), so even queries whose individual segments compile quickly may have multiple seconds of total compilation latency.

2.3 Limitations of Existing Solutions

Several mitigations exist, summarized in Table 3, but none eliminates the cold-start problem. One natural response is to maintain separate interpretation and compilation engines, as some database systems do. However, this introduces substantial engineering complexity: duplicate operator implementations must be maintained, debugged, and optimized independently, and ensuring semantic equivalence between code paths requires extensive testing.

Table 3: Existing mitigations for compilation cold-start.

Mitigation	**Why it falls short**
Separate engines	Requires a custom IR, a custom compiler backend, and switching logic. Both backends still generate operator *logic* at query time. Compilation is faster, but not eliminated [16, 24].
Compilation caching	Local caches are invalidated by upgrades and evicted under memory pressure. Predicate changes and adaptive optimization produce new plans that miss even the remote cache [8].
Serverless compilation	Parallelizes compilation for 2–3× speedup [8], but a 3 s compilation reduced to 1.5 s still dominates a 100 ms query.
Constant parameterization	Reuses compiled code when only predicate values change, but a different number of predicates produces different code. Even with the same predicates, selectivity changes can reorder evaluation, producing new code paths. Also removes the opportunity for certain compiler optimizations.

Recent systems like Umbra [16, 24] mitigate cold-start latency with a fast custom backend for immediate execution, but both backends still generate operator *logic* at query time (§8 provides a detailed comparison).

Compilation caching and serverless compilation [8] reduce but do not eliminate this overhead. Redshift's two-tier code cache (local per-node, remote shared [8]) achieves a hit rate above 99.9% in steady state, but this understates the cold-start problem: software upgrades clear local caches, adaptive optimization produces new plan shapes, and predicate variations generate previously unseen segments. Fleet-wide analysis [32] reveals severe hour-to-hour spikes in query load that introduce new patterns missing even the remote cache.

This is particularly acute for interactive dashboards, applications where users explore data by selecting filters from dropdown menus, each selection triggering a new analytical query. A user selecting a different value (a different artist, course, time range, or category) can change predicate selectivity enough to alter the optimizer's join ordering or access path, producing a different plan and missing the code cache. An otherwise sub-3-second dashboard refresh then spikes to 8–12 seconds while compilation completes, severely degrading the interactive analytics experience. ETL pipelines suffer

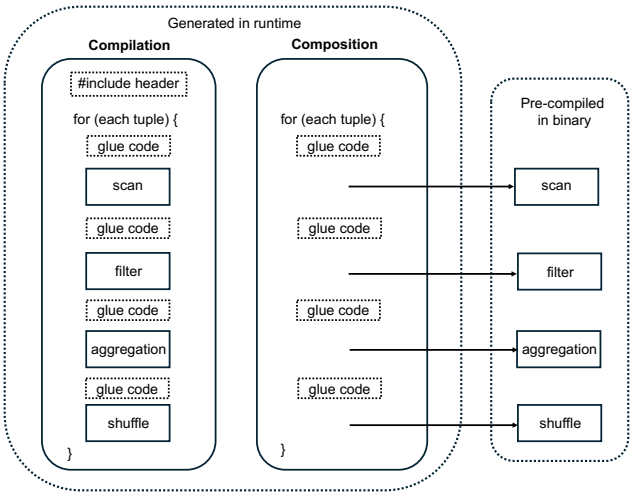

Figure 1: Compilation (left) vs composition (right).

similarly: their mix of `INSERT`, `UPDATE`, and `DELETE` statements produces many unique plans, and compilation overhead accumulates across the entire run.

Parameterizing predicate values mitigates but does not solve the issue: different numbers of predicates produce different code, and selectivity changes can reorder evaluation, producing new code paths. The combination of frequent releases and adaptive execution means that some fraction of the fleet is always warming up.

3 Composition

Composition addresses the cold-start problem by changing *what* is generated at query time. Unlike compilation, which generates operator logic that must be compiled, composition generates only the arrangement of pre-existing logic: operator implementations already compiled into the binary at build time. At query time, the system generates lightweight glue code (function calls, stack allocations, and control flow) that wires operators and expressions together. This glue code is compiled via LLVM into a shared library in milliseconds, avoiding unbounded optimization at query time while retaining the execution benefits of compiled code. Figure 1 illustrates the distinction.

Composition enables immediate execution while compilation proceeds in the background. The fully optimized compiled object is cached and serves subsequent executions of the same segment for peak performance. The technique shares conceptual roots with threaded code [5] and other prior techniques; Section 8 discusses the key differences.

Composition addresses both cost categories identified in §2.2. It avoids frontend costs because it never generates C++ source code: there are no headers to parse and no templates to instantiate. It avoids backend costs because the glue code contains only function calls through opaque pointers, with no operator bodies to optimize. The result is that cold queries start executing in milliseconds instead of waiting seconds, closing the latency gap between cold and warm queries.

This is because the generated glue code is deliberately simple: it declares stack variables, calls pre-compiled operator functions, and arranges control flow (loops, branches) among these calls. The operator implementations are hidden behind opaque pointers, so the compiler cannot attempt to inline or optimize across operator boundaries, bounding compilation cost to scale with operator count rather than superlinearly with code size as in full compilation. Each operator contributes a fixed number of function calls and allocations to the glue code, with no cross-operator expansion. Because the glue code is linked into a separate shared library that calls into the main engine binary at runtime, cross-boundary optimization is structurally impossible. This code is so straightforward that one could emit native machine code for it directly. However, using an existing compiler backend achieves the same result with less engineering effort. Composition exploits the observation from §2.2 that hiding operators behind opaque pointers, normally incompatible with cross-boundary compiler optimizations, becomes viable as a temporary execution mode while compilation proceeds in the background.

Both compiled and composed code execute fully optimized operators. The difference is that compilation additionally optimizes across operator boundaries, while composition preserves boundaries as function calls. The glue code itself is compiled: loop control, increments, and control flow are optimized by LLVM, and call targets are statically determined. There is no interpretive dispatch loop. Moreover, in practice not all parts of a query need composition: portions served from cached compiled code retain full performance, so the composition penalty is diluted at the query level.

3.1 Design Goals

To deploy composition in a production query execution system, three goals must be met.

Eliminate cold-start overhead with bounded cost. Composition must enable near-instant start of execution for cold queries by restricting query-time work to linking and arranging pre-optimized operator modules rather than optimizing a fused pipeline from source. Execution time under composition should remain comparable to fully compiled execution, so that the performance gap is limited to one-time composition overhead rather than sustained per-tuple cost. Without this, interactive dashboards and latency-sensitive pipelines cannot tolerate cold runs.

Single codebase. The same operator implementations must serve both composition and compilation modes, avoiding duplicate code and ensuring semantic equivalence (§4). Without this, every operator change requires updating two implementations, doubling development effort, slowing feature velocity, and introducing correctness bugs when the two paths diverge.

Transparent coexistence. The system must automatically choose between composition (cold) and compilation (cached) with no user intervention, and support hybrid execution mixing both within a single query plan. Without this, operators or users must manually manage execution modes, adding complexity and risk.

4 Composition in Amazon Redshift

This section describes how we meet the aforementioned goals in Amazon Redshift through FastCompose.

(a) Query (Q3)

```
SELECT
    l_orderkey,
    SUM(l_extendedprice *
        (1 - l_discount))
        AS revenue
FROM
    customer,
    orders,
    lineitem
WHERE
    c_mktsegment = 'BUILDING'
    AND c_custkey = o_custkey
    AND l_orderkey = o_orderkey
    AND o_orderdate < '1995-03-15'
GROUP BY
    l_orderkey
ORDER BY
    revenue DESC
```

(b) Query Plan

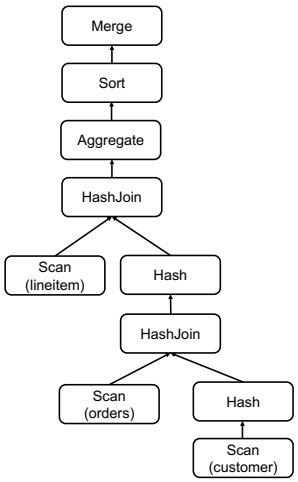

Figure 2: Running example: TPC-H Q3.

4.1 Code Generation and Compilation

Amazon Redshift [2, 14] is a column-oriented, massively parallel data warehouse. The leader node parses, optimizes, and generates executable code for queries. The compiled code is sent to the compute nodes which execute it on the data they are assigned to process.

Redshift's execution engine divides each query plan into *segments*, pipeline units bounded by materialization (hash builds, sorts) or data redistribution (shuffles, broadcasts). Each segment comprises a sequence of *steps*: operators such as scans, filters, projections, joins, and aggregations. For each segment, the leader generates a C++ source file containing a fused function that combines all steps, annotated with `always_inline` to direct g++ to optimize aggressively. The resulting shared library is broadcast to compute nodes for execution.

Redshift caches compiled objects at two levels to amortize the cost of compilation: a local cache on each node and a remote cache backed by a serverless compilation service [8, 26]. On a cache miss, the query waits for g++ compilation, the cold-start problem that composition addresses.

4.2 FastCompose Overview

We use a simplified TPC-H Q3 depicted in Figure 2 as a running example. Amazon Redshift's execution engine produces seven segments for this query based on the query plan provided by the optimizer (Figure 2, panel b), separated by network redistributions and hash table materializations: (1) scan and filter `customer`, shuffle by join key; (2) receive shuffled tuples, build hash table; (3) scan and filter `orders`, shuffle by join key; (4) receive shuffled tuples, probe hash table built with `customer`, shuffle joined results by next join key; (5) receive shuffled tuples, build hash table; (6) scan `lineitem`, probe hash tables, project, aggregate, sort, and return results to the leader; (7) receive sorted results from compute nodes, perform final merge, return to client.

In compilation mode, each segment is independently compiled into a separate dynamic library by g++. Execution proceeds as dataflow dependencies are satisfied: a segment begins executing as soon as its compiled object is available and its input data is ready. Nevertheless, each cache miss adds seconds of compilation latency to the segment's hot path. In composition mode, the Composer generates glue code for each segment that connects operator implementations already in the binary, reducing per-segment compilation from seconds to milliseconds.

Three build-time mechanisms enable composition from the same codebase used for compilation: the *Injector* transforms existing code generation logic to support both modes (§4.4), the *Composer* DSL generates glue code at query time (§4.5), and the *Shielder* hides operator internals behind stable interfaces so that composition never needs to process operator definitions at query time (§4.6). The end-to-end execution flow, including the cache lookup and leader-to-compute-node broadcast, is described in Section 4.7.

4.3 Unified Operator Implementation

The central design goal is that developers write each operator *once*, and the system automatically produces both the compilation output (fully optimized) and the composition output (immediate execution). This subsection explains how a single operator implementation serves both modes.

Each operator in Redshift is implemented as a *step* with a code generation method called `generate_code()`. A driver loops over the steps in a segment and calls `generate_code()` on each one. Inside this method, the developer uses API calls (`put_stmt()` to emit a code statement, `put_blok()` to open a block such as a loop or conditional, and `put_endr()` to close it) to describe the operator's logic. The emitted code is organized into phases (including setup, per-row processing, and teardown, among others) that the driver assembles into the final fused function for the segment.

For example, consider the sort step that implements Q3's `ORDER BY` clause. It emits code to construct a sorter (setup), add each aggregated record (per-row), and sort the accumulated records (teardown). Figure 3 shows three representations of this step side by side: the source code that the developer writes (panel a), the output for compilation mode (panel b), and the output for composition mode (panel c). The bottom row contrasts the resulting machine code: compiled code contains hundreds of inlined instructions with operator boundaries eliminated, while composed code reduces to a few function calls per operator. The *Injector* (§4.4) makes this possible by transforming each operator's code generation logic at build time.

4.4 The Injector: Code Preprocessing

The challenge in supporting both modes from a single codebase is that the code generation calls (e.g., "emit a sort operator") produce text strings with no type information. The system knows *what* code to emit but not the types, sizes, or signatures of the operators involved. This information is needed to generate the composition mode function calls. Parsing the operator definitions at query time would be as expensive as compilation itself, defeating the purpose.

The Injector solves this by performing the analysis once at *build time*. It is a preprocessor that transforms each operator's code generation method to emit both compilation and composition output. It runs once when the Redshift binary is built, not at query time.

The Injector operates in two passes. In the first pass, it scans each step source file, locates all `put_stmt`, `put_blok`, and `put_endr` calls,

(a) Source: `put_stmt` calls

```
// Setup
put_stmt(
 "StableSorter<%s*,%s>"
 " sorter",
 rec_type, cmp_func);
put_stmt(
 "sort_step->"
 "set_sorter(&sorter)");

// Per-row
put_stmt(
 "sorter.add_record"
 "(%s)", record);

// Teardown
put_stmt(
 "sorter.sort()");
```

(b) Compilation mode (C++ for g++)

```
// Setup
StableSorter<SortRec0*,
  Comparator0> sorter;
sort_step->
  set_sorter(&sorter);

// Per-row
sorter.add_record(
  sort_rec);

// Teardown
sorter.sort();

// g++ inlines all calls,
// eliminates operator
// boundaries, optimizes
// the fused pipeline.
```

(c) Composition mode (glue code)

```
// Setup
char sorter[SIZEOF_SS];
ConstructStableSorter(
  &sorter, comparator0);
sort_step_set_sorter(
  sort_step, &sorter);

// Per-row
sorter_add_record(
  &sorter, sort_rec);

// Teardown
sorter_sort(&sorter);

// Compiler sees only
// function signatures and
// buffer sizes.
```

(d) After Injector (setup)

```
// Original:
put_stmt(
 "StableSorter<%s*,%s>"
 " sorter",
 rec_type, cmp_func);

// Injected:
if (compose_enabled) {
  CreateObject<
    StableSorter>(
    Ident("sorter"),
    rec_type, cmp_func);
}
```

(e) Compiled assembly (g++ -O3)

```
; Inlined add_record:
; operator boundaries
; eliminated, loop body
; fused with comparator
mov  rdi, [rbp-0x28]
movq xmm0, [rsi+0x10]
cmp  eax, [rdi+0x08]
jge  .resize
mov  [rdx], rax
movq [rdx+0x08], xmm0
add  qword [rdi+0x18], 1
; ... (100s of inlined
;     instructions)
```

(f) Composed assembly

```
; add_record call:
; just load args and call
lea  rdi, [rbp-0x120]
mov  rsi, r14
call sorter_add_record

; sort call:
lea  rdi, [rbp-0x120]
call sorter_sort

; Total: ~5 instructions
; per operator call.
; No inlined bodies.
```

Figure 3: Composition reduces per-operator code from hundreds of inlined instructions to a few function calls, explaining why it compiles in milliseconds instead of seconds. (a) The developer writes `put_stmt` calls once. **(b)** Compilation mode produces C++ that g++ compiles with full inlining (seconds). **(c)** Composition mode produces glue code that calls the same implementations through function pointers (milliseconds). **(d)** After the Injector, both the original `put_stmt` and the injected Composer DSL call coexist in a single code path. **(e)** Compiled assembly: hundreds of inlined instructions. **(f)** Composed assembly: a few `call` instructions per operator.

and parses their printf-style format strings to extract variable names and operations, information needed to generate the corresponding Composer DSL calls. In the second pass, it injects Composer DSL calls alongside each original `put_*` call. The original calls are preserved unchanged, so compilation mode continues to work identically.

The Injector does not require a C++ parser. Because `put_stmt` format strings follow a constrained set of patterns—variable declarations, assignments, method calls—regular-expression matching suffices to transform them into the corresponding Composer DSL expressions. These DSL expressions are needed because composition mode generates LLVM IR through typed function calls rather than text strings. Where the format strings alone are insufficient—for example, to determine that `StableSorter` is a class template requiring object construction rather than a simple variable declaration—developers annotate the code with lightweight directives that supply the missing type and semantic information. For instance, a directive declares that `sorter` is an object of type `StableSorter`, enabling the Injector to generate the correct object-construction call in the composition output using the Composer DSL.

Correctness is enforced at build time. The injected Composer DSL calls are themselves C++, so the compiler type-checks them during the Redshift build time, catching missing directives, type mismatches, and interface errors before any query executes. The developer provides the directives, and the compiler enforces them.

Semantic Equivalence. Why are the compilation and composition outputs semantically equivalent? Because both are generated from the *same semantic source*: the `put_stmt` calls in each step's `generate_code()` method. The Injector does not introduce a separate operator implementation. Instead, it derives the composition code path from the same `put_stmt` calls that define the compilation path. This is why we describe it as a parallel *code path* rather than a parallel *implementation*: the developer writes operator logic once, and the Injector generates the corresponding DSL calls mechanically.

At query time, the driver invokes `generate_code()` once, and both paths execute. The generated content determines the cache key, so code generation always runs (§4.7): the original `put_stmt` calls emit C++ text for compilation mode, while the injected DSL calls emit LLVM IR glue code for composition mode. The two outputs are semantically identical: they invoke the same operator implementations with the same arguments in the same order. The only difference is the calling convention: compilation inlines for maximum performance, composition calls through opaque pointers for bounded compile time.

Because both code paths are mechanically derived from the same `put_stmt` calls, any operator change automatically propagates to both modes. The semantic correctness of the developer-supplied directives is validated through build-time type checking and testing (§7).

Figure 3(d) illustrates this: the original code generation call and the injected Composer DSL call coexist side by side. Which output is used for execution depends on the cache lookup (§4.7).

4.5 Composer: An Embedded DSL for Glue Code

The Composer is the component that actually generates the composition mode output. It is an embedded DSL (domain-specific language) that mirrors the structure of the code generation calls but produces lightweight function calls instead of C++ source text. Where the compilation path emits "create a sorter of this type," the Composer emits "call the pre-compiled sorter constructor with these arguments."

The Injector transforms the existing code generation calls into Composer DSL calls (§4.4). The DSL is embedded in C++ and mirrors C++ constructs but generates a compiler intermediate representation (LLVM IR in our implementation) instead of source text. It provides typed abstractions for variables (`CreateVar<T>`), objects (`CreateObject<T>`), arrays (`CreateArray<T>`), function calls, assignments, control flow (if, while, for, switch), and scoping—each generating the corresponding IR instructions.

The DSL is designed to map naturally to the patterns found in `put_stmt` calls. Where a `put_stmt` emits a `StableSorter` declaration, the corresponding Composer call is `CreateObject<StableSorter>(Ident("sorter"), ...)`. Where a `put_blok` emits an `if` block, the Composer equivalent is `COMPOSE_IF(...)`. This close correspondence is what makes the Injector's build-time regular-expression-based transformation feasible: each `put_*` pattern has a direct Composer counterpart.

A property of the DSL is that operator hiding is built in. When the Composer creates an operator object, it automatically uses the wrapper interface generated by the Shielder (§4.6), allocating a buffer of the correct size and calling the constructor through a function pointer. The developer never writes wrapper code. The DSL handles it.

One subtlety arises from how database engines generate code: operators emit statements into different phases (setup, per-row, teardown) in whatever order is convenient, not in execution order. A variable might be used in the per-row phase before its declaration is added in the setup phase. Composer handles this through lazy resolution: statements can be added to any phase out of order, and variable types and class layouts are finalized only when the phases are assembled into their final linear order and the IR module is created.

4.6 Shielder: Hiding Operator Internals

For composition to be fast, the compiler must not see the internal details of operator implementations. Otherwise it would attempt to parse, analyze, and optimize them, reintroducing the compilation cost. The Shielder addresses this using the pImpl (pointer to implementation) pattern in C++.

Without the Shielder, the compiler would need to see the full class definitions (constructors, method bodies, template specializations) to generate correct code for operator calls. These definitions live in Redshift's headers, and the precompiled header used by the code generation subsystem is large. Parsing these headers at query time would be as expensive as g++ compilation, defeating the purpose of composition.

FastCompose avoids this through automatic interface generation. The Shielder scans operator class headers for lightweight annotations and generates an interface for each annotated class. The generated interface exposes only what the Composer needs: the size of the operator object (for buffer allocation), static helpers for the constructor and each method (which take an opaque pointer and forward the call), and member offsets (so fields can be accessed via pointer arithmetic without knowing the object layout).

The IR generated by the Composer sees only these interfaces—function signatures, variable sizes and buffer sizes—never the operator internals. This is what bounds composition's compilation cost: the composed code is a thin layer of function calls and buffer allocations, with no operator definitions to parse, no templates to instantiate, and no function bodies to optimize.

Table 4 summarizes how the three mechanisms address the compilation costs identified in Section 2.2.

Table 4: How composition addresses compilation costs.

Cost	How composition addresses it
Frontend (parsing, templates)	The *Shielder* hides operator internals behind stable interfaces, avoiding the need to parse operator definitions at query time. The *Injector* generates Composer DSL calls, guided by developer-provided directives, that reference pre-compiled operators using explicitly instantiated templates compiled into the binary at build time.
Backend (unbounded optimization)	The *Composer DSL* generates only function calls, stack allocations, and control flow among high-level operator calls. LLVM optimizes only this thin layer. Operator bodies remain behind opaque pointers and are never exposed to the optimizer.

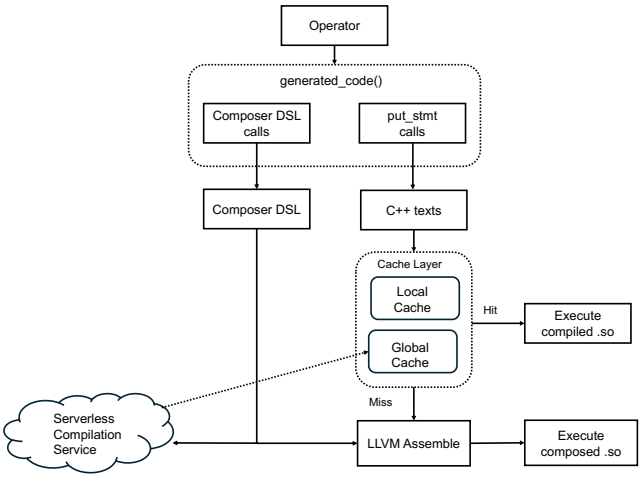

Figure 4: Query-time code generation flow for a single segment. `generate_code()` always produces both C++ text and Composer IR in a single invocation. On a cache hit, the cached shared library executes directly. On a miss, the Composer IR is assembled into a shared library for immediate execution while g++ compiles the C++ text in the background. The compiled object is cached and used by subsequent executions of the same segment.

4.7 Execution Flow

This subsection describes what happens at query time when a segment needs to execute: how the system checks the cache, decides between composition and compilation, and delivers the executable code to compute nodes (Figure 4).

For each segment in a query plan, the leader node runs the step's `generate_code()` method, which always produces both the C++ text (via `put_stmt` calls) and the Composer IR (via the injected DSL calls). The generated content determines the cache key. On a cache hit, the cached shared library (compiled or composed) is broadcast to compute nodes. On a miss, the Composer IR is compiled via LLVM into a shared library on the leader and broadcast immediately, while the C++ text is submitted for serverless compilation. The compiled library replaces the composed one when ready. Compute nodes are unaware of which mode produced the library.

4.8 Mixing Execution Modes

Composition and compilation are not all-or-nothing per segment. By default, all operators are invoked through pImpl interfaces. For simple operators where call overhead is measurable (e.g., `sum()` aggregation), the developer can selectively expose the implementation via direct memory access. For example, for setters and getters, we can directly implement them using Shielder-derived offsets (e.g., `*(TYPE*)(ptr + offset)`), keeping LLVM compilation fast while removing call overhead on the hot path. Similarly, scan-layer operations use pre-compiled SIMD-optimized functions that both composed and compiled paths call identically. Composition naturally supports this hybrid model.

5 Implementation

This section presents implementation details: operator coverage, the developer annotation effort, build system integration, and engineering decisions made during deployment.

Operator coverage. If any segment cannot be composed, it must wait for compilation, reintroducing the cold-start latency that composition eliminates. In the current codebase, composition covers all production query execution paths; the remaining unsupported paths represent less than 0.5% of all code generation calls and are confined to testing and debugging infrastructure that is never exercised by production queries.

Directive burden. Developers annotate operator source files with directives (C++ macros prefixed with `COMPOSE_`) that supply the type and semantic information the Injector needs at build time. In the current codebase, roughly one directive is needed for every two code generation calls (`put_*`). The directive count is inflated because directives must be present at both declaration and use sites. The same variable appearing across multiple source files artificially increases the number of directives needed.

Two kinds of annotations are used. *Injector directives* declare variable types and objects (`COMPOSE_DECL`) or provide pattern-matching rules that map a `put_*` format string to a sequence of Composer DSL instructions. For example, given:

```
COMPOSE_DECL(SortStep*, step);
COMPOSE_DECL(QueryPlan*, plan);
put_stmt("SortStep* step = plan->GetSortStep()");
```

the Injector emits the original `put_stmt` alongside a guarded Composer DSL call:

```
put_stmt("SortStep* step = plan->GetSortStep()");
if (composition_mode_enabled) {
  CreateVar<SortStep*>("step") =
      LookupVar<QueryPlan*>("plan")->GetSortStep();
}
```

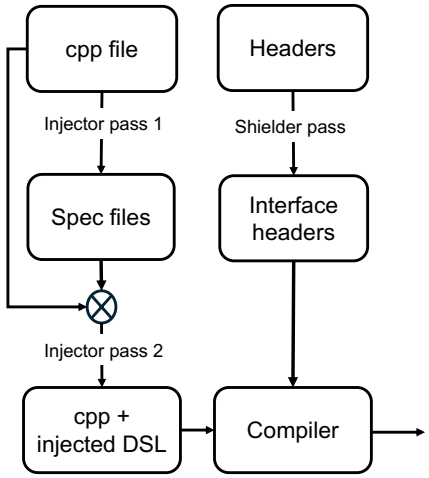

Figure 5: Build-time pipeline.

Shielder annotations mark operator classes in header files so the Shielder can generate pImpl interfaces. Developers annotate classes with `COMPOSE_CLASS` and methods with `COMPOSE_METHOD`:

```
COMPOSE_CLASS
class QueryPlan {
  COMPOSE_METHOD
  SortStep* GetSortStep();
};
```

From these annotations the Shielder generates an interface for each annotated class. For every annotated method, it emits a static helper that takes an opaque pointer to the object plus the method's arguments and forwards the call, e.g., `GetSortStep` above becomes a helper that accepts a raw pointer to the `QueryPlan` and invokes the real method. The interface method emits Composer DSL that calls this helper at runtime. Because the glue code only ever calls through static helpers with opaque pointers, it never needs to know the type or layout of the underlying operator.

Build system integration. The Injector runs as a two-pass build step before g++ compilation, producing a generated source file for each step file. The Shielder separately scans annotated class headers and produces pImpl interface files. Both tools run at build time with no per-query overhead.

Object file generation. At query time, the Composer produces LLVM IR for the glue code. LLVM's code generation backend converts this IR into machine code, which is linked into a dynamic library on the leader node and broadcast to compute nodes through the same code cache infrastructure used for compiled objects. An earlier design instead compiled the IR to machine code in memory on each compute node and executed it directly (JIT compilation), but we found the dynamic library approach superior for three reasons. First, JIT compilation would consume CPU and memory on every compute node, whereas generating a shared library on the leader centralizes this cost. Second, Redshift's compute nodes periodically restart worker processes, discarding in-memory JIT code. Dynamic libraries persist across restarts. Third, the shared library integrates naturally with

Redshift's existing code cache and three-tier caching infrastructure—composed libraries are cached, served to subsequent queries, and replaced by compiled objects from the remote cache when ready, all without changes to the compute node code path. While LLVM's ORC JIT framework supports remote execution across processes, adopting it would require building additional infrastructure for cross-node memory management and communication, whereas the dynamic library approach reuses Redshift's existing code distribution path.

Expression code generation. Redshift's existing expression infrastructure generated C++ text fragments that relied on type inference (`auto result = f0 + f1`), bypassing the `put_stmt` infrastructure. This worked for g++ but not for composition, which requires explicit types. We built a new expression code generation infrastructure driven by PostgreSQL's function catalog (`pg_proc.h`), which provides type signatures for all built-in functions and operators. From this catalog, the system automatically generates both the C++ text for compilation and the Composer DSL calls for composition, ensuring type-correct code for all expression types without manual annotation.

Expression composition. The binary contains pre-compiled scalar functions (e.g., `btrim`, `lower`, `concat`). At query time, glue code calls these and threads results between them: each function's output is stored in a local variable and passed as an argument to the next. A nested expression like `concat(lower(trim(x)), ...)` becomes a linear sequence of calls. To bound composition time, the Composer splits complex expressions into sub-functions, each containing a bounded number of calls. LLVM's inlining then selectively inlines across sub-function boundaries, capping compile time while removing per-call overhead for short expression chains.

6 Evaluation

We evaluate FastCompose on standard benchmarks, production fleet data, and customer workloads.

6.1 Benchmark Performance

We use TPC-DS and TPC-H at 100 GB and 3 TB scale factors on 48-RPU (Redshift Processing Units) serverless workgroups. Each workload is measured in three configurations: (1) *Cold Compilation*: compilation-only baseline with empty code cache and serverless compilation service active; (2) *Cold FastCompose*: FastCompose enabled with empty code cache and serverless compilation service active; (3) *Warm Compilation*: fully compiled code served from cache, representing steady-state performance. For cold configurations, the code cache is effectively invalidated so that all segments compile on first encounter, while repeated segments within the same run still benefit from caching.

Table 5 summarizes end-to-end workload performance across benchmarks and instance types. All runtimes are normalized to Warm Compilation (steady state).

On compilation-dominated workloads (100 GB), the cold-start penalty is severe: Cold Compilation is 26.3× (TPC-DS) and 39.1× (TPC-H) slower than steady state. FastCompose reduces this to 3.8× and 3.3× respectively: speedups of 7.0× and 12.0×. At smaller scales, speedups are 10.5×/8.7× on TPC-H SF1/SF10 and 4.7×/4.5× on TPC-DS SF1/SF10, slightly below the 100 GB peak because composition's fixed overhead is a meaningful fraction of very short query runtimes.

Table 5: Benchmark performance normalized to Warm Compilation, and per-query speedup distribution (Cold Compilation / Cold FastCompose).

Workload	Cold Comp.	Cold FC	Speedup	Min	Med.	Max
TPC-DS 100G	26.3×	3.8×	7.0×	1.6×	7.2×	21.7×
TPC-DS 3T	2.2×	1.1×	2.0×	1.0×	2.9×	10.3×
TPC-H 100G	39.1×	3.3×	12.0×	4.1×	11.9×	30.0×
TPC-H 3T	1.8×	1.1×	1.6×	1.0×	1.8×	4.4×

At 3 TB scale, execution time dominates and compilation overhead is amortized. FastCompose still achieves 2.0× speedup on TPC-DS and 1.6× on TPC-H, with composed execution within 10% of fully compiled execution. We observe consistent results on provisioned instances (ra3.4xl, ra3.xlplus), confirming that composition scales across cluster configurations. The 2–10% gap in execution time reflects the full three-tiered cache configuration: a cold query executes a mix of composed segments (local cache misses) and compiled segments (cache hits from earlier queries in the same run), not a purely composed execution. Two patterns account for most of the remaining gap: (1) nested per-tuple expression calls (e.g., `concat(lower(trim(x)), ...)`), where compilation inlines the entire call chain, and (2) tight loops where type-specialized (templatized) code lets the compiler fuse operators across boundaries. For (1), composition mitigates the overhead by strategically inlining the glue code for nested expression calls, narrowing the gap in practice. Composition eliminates the cold-start stall on the critical path, while compiled segments already serve the cached parts of the plan.

Table 5 also shows per-query speedup distributions and Figure 6 shows the per-query detail. At 100 GB, the highest speedups occur on compilation-dominated queries: TPC-H Q6 achieves 30.0× (10.1s → 0.3s) and TPC-DS Q44 achieves 21.7× (30.7s → 1.4s). At 3 TB, speedups are smaller but still meaningful (median 2.9× on TPC-DS, 1.8× on TPC-H).

The pattern reflects the tandem design: at 100 GB queries finish before compiled objects arrive, so all segments run composed; at 3 TB queries run long enough that the compilation service delivers compiled objects for later segments in the same query. To isolate this effect, a compose-only configuration (no compiled objects used at all) is 12% slower than standard FastCompose on TPC-DS 100 GB, 22% on TPC-DS 3 TB, and 63% on TPC-H 100 GB, confirming that compiled objects progressively improve performance within a single run.

6.2 Composition vs. Compilation Time

Table 6 compares composition time against g++ compilation time across percentiles from production workloads running on the Redshift fleet.

Composition completes in 16–49 ms at p0–p25 and 91 ms at the median, versus 2.4–5.5 s for g++ (61× at p50). The speedup is highest at low percentiles where composition's fixed overhead dominates and g++ still pays for parsing and template instantiation. At higher percentiles both times grow with segment complexity, but composition grows more slowly because it generates only function calls and control flow. The speedup is non-monotonic (65× at p90 vs. 48× at p75) because compilation jumps sharply at complexity thresholds

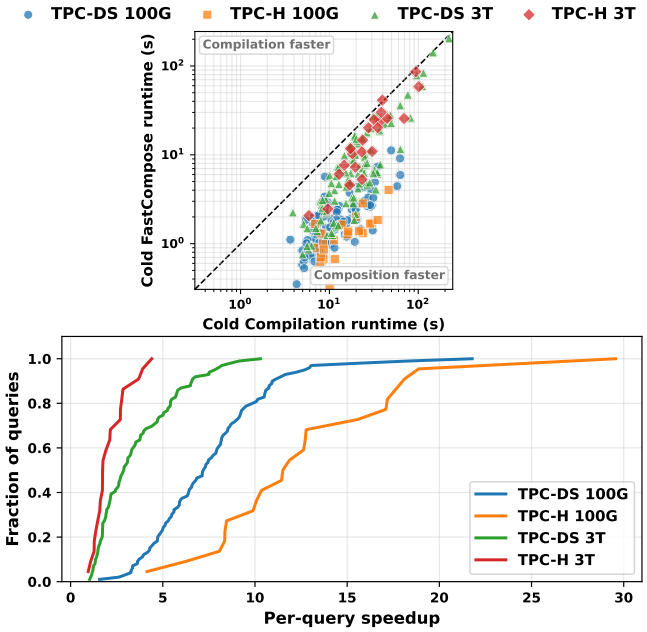

Figure 6: Per-query analysis. Top: Cold Compilation vs. Cold FastCompose runtime. Bottom: CDF of per-query speedups.

Table 6: Composition vs. compilation time (ms) across production workloads.

Percentile	Composition	Compilation	Speedup
p0	16	2,445	153×
p25	49	4,619	94×
p50	91	5,554	61×
p75	203	9,720	48×
p90	547	35,651	65×
p95	1,136	37,906	33×
p99	6,477	76,005	12×
p99.9	21,461	158,173	7.4×

Table 7: Generated object size (KB) across the production fleet.

	p25	p50	p75	p90	p99
Composed	30	43	75	86	167
Compiled	132	230	947	1,231	1,998
Ratio	**4.4×**	**5.4×**	**13×**	**14×**	**12×**

while composition grows smoothly. The tail narrows (7.4× at p99.9) because the largest segments produce substantial glue code; composition at high percentiles also includes leader-node queuing delays.

Object size. Table 7 compares generated shared library sizes. Composed objects contain only glue code; compiled objects embed operator logic.

At the median, composed objects are 5.4× smaller (43 KB vs. 230 KB), and 14× smaller at p90 (86 KB vs. 1.2 MB), confirming that composed objects scale with operator count, not implementation complexity.

Table 8: Enterprise dashboard workload (363K queries, ra3.4xl).

Configuration	Total Duration	Execution Time
Warm Compilation	1.0×	1.0×
Cold Compilation	4.35×	1.24×
Cold FastCompose	1.25×	1.02×
Speedup (Cold Compilation / Cold FastCompose)	**3.49×**	

Table 9: Enterprise ETL workload (4.3K queries, ra3.16xl).

Query Type	Duration Speedup	Compile Speedup
`INSERT` (183)	2.74×	6.1×
`UPDATE` (728)	1.74×	3.5×
`SELECT` (3,134)	3.83×	6.9×
`DELETE` (6)	6.93×	10.7×
Overall (4,305)	**1.93×**	**3.8×**

6.3 Customer Workloads

We evaluated FastCompose on production-representative workloads from large enterprise customers.

Dashboard Workload. The workload replays 363K queries on a 4-node ra3.4xl cluster, exercising join-heavy analytics with dropdown-driven predicate variations (the pattern described in §1). Table 8 summarizes the results, normalized to Warm Compilation (fully compiled, cached).

Without FastCompose, the cold run takes 4.35× longer than steady state, dominated by compilation overhead even though execution is only 1.24× warm. With FastCompose, the cold run lands within 25% of steady state (1.25×): a 3.49× speedup. Composed execution stays within 2% of fully compiled performance (1.02×); the remaining 25% in total duration is composition time for segments not yet replaced by compiled objects from the remote cache.

ETL workload. We also evaluated an enterprise ETL pipeline of 4,305 queries mixing `INSERT`, `UPDATE`, `DELETE`, and `SELECT` statements with complex expressions and distinct aggregations, run on a 16-node ra3.16xlarge cluster. No warm baseline is available, so we compare Cold Baseline (compilation) against Cold FastCompose directly.

Table 9 shows the results by query type. `INSERT` queries, the dominant cost, see a 2.74× reduction in total duration and a 6.1× reduction in compilation time. `UPDATE` queries improve by 1.74× overall with 3.5× less compilation. `SELECT` queries, though lightweight individually, benefit the most (3.8× duration, 6.9× compilation) because their short execution times make compilation overhead proportionally larger. Across all query types, FastCompose reduces total compilation time from 13,803s to 3,562s, a 3.8× improvement.

A secondary benefit: large ETL segments force the local compiler to lower optimization levels to stay within resource budgets, so the entire generated code—operator logic included—runs with reduced optimization until the serverless compilation service delivers a fully optimized build. With FastCompose, operator code is pre-compiled at build time with full optimizations, so composed queries execute

with fully optimized operators immediately regardless of segment complexity.

7 Experience

7.1 Development Journey

Before FastCompose, Redshift addressed cold-starts with a separate pre-compiled execution engine that implemented operators independently from the compilation path. It provided temporary relief but doubled development effort, exhibited divergent behavior across the two engines, and supported only a subset of operators—queries touching unsupported operators fell back to compilation, producing unpredictable latency spikes. This motivated composition: removing cold-start latency from a *single* codebase with semantic equivalence guaranteed by construction (§4).

7.2 Testing and Validation

Validating semantic equivalence requires running both paths and comparing results. We alternate configurations across nightly runs (composition-only one night, compilation-only the next), keeping the resource budget unchanged while covering both paths. Customer workloads run alongside standard benchmarks during pre-production testing and surfaced issues that synthetic benchmarks did not.

7.3 Deployment Experience

Staged rollout. The new expression code generator (§5) was a hard dependency for FastCompose, so we deployed it first without enabling composition. The two-phase rollout isolated expression-generator issues from composition issues. Within each phase, deployment was staggered cluster by cluster, validating performance before expanding.

Fleet monitoring and controls. Fleet telemetry surfaces performance anomalies early. For example, an expression-generator optimization that did not apply on certain constant expressions was caught before regional rollout. Both the expression generator optimizations and FastCompose itself can be disabled at fine granularity (per cluster, per query pattern, or per optimization). When a semantically incorrect directive caused a runtime error, we disabled composition for the affected paths without affecting other queries.

Customer impact. Across customer engagements in finance, telecom, and manufacturing, dashboard workloads with strict latency SLAs saw cold-run duration drop by 3–3.5×. The cold/warm gap that previously forced operational workarounds disappears in practice: one enterprise had run a dedicated standby cluster solely to pre-populate the code cache before production hours, and retired it after enabling FastCompose. A financial services customer cited FastCompose as the key enabler for meeting their reporting SLA across nine production clusters where plan changes had previously caused unpredictable spikes.

7.4 Maintenance

Developer onboarding. User manuals, how-to guides, and AI-assisted tooling that suggests directives from existing patterns keep onboarding smooth. New Composer DSL APIs are added on demand; this has happened once to date.

Error diagnostics. The Injector delegates type checking to g++, so directive errors surface as template errors that can be hard to trace back to the source directive. Improving the Injector's own error reporting is ongoing.

7.5 Performance Lessons

Dynamic libraries over JIT. The JIT vs. dynamic library trade-off (§5) became apparent only under high-concurrency production workloads. TPC-DS and TPC-H did not surface it.

Optimization levels. IR-level `-O2` matters: passes such as `mem2reg`, `simplifycfg`, and `instcombine` substantially reduce the unoptimized IR the Composer produces, lowering overall composition time. Machine-level optimization adds significant compile time (slower instruction selection and register allocation) with no measurable execution-time improvement for glue code, which is mostly function calls and control flow. We therefore always use `-O2` at IR level, and default to `-O2` at the machine level but lower to `-O0` adaptively as segment size grows.

Large segment regression. On segments wider than 1,000 columns, composition was initially *slower* than g++. Redshift already detects resource-intensive g++ compilation and falls back to lower optimization locally while submitting a fully optimized build to the serverless compilation service; composition lacked this adaptation. Applying the same model resolved the regression.

LLVM over a custom backend. A direct IR-to-machine-code backend would reduce composition time further but requires building and maintaining a custom compiler. LLVM provides a stable foundation that benefits from community improvements, which we prioritize for a production system serving thousands of clusters.

Benchmarks vs. production. TPC-DS and TPC-H did not surface several of the issues above. The large-segment regression was triggered by a customer workload that updated a 1,000-column table. These experiences reinforce the importance of testing with real customer workloads alongside standard benchmarks.

Applicability beyond Redshift. The technique applies to any compiling engine whose code generation layer emits textual code through a structured API. A build-time preprocessor extracts type information from the generation calls and injects a parallel composition path. Retaining the text-based interface keeps debugging straightforward because the generated text is readable, so the barrier to adoption is directive annotation, not a new programming model.

7.6 Limitations and Future Work

Glue code size grows with query complexity. Segments with very wide projections, deeply nested joins, or many distinct aggregations produce proportionally more glue code, increasing composition time. Queries with many segments incur elevated overhead on the first few, though the three-tier cache mitigates this as later segments use compiled code from the remote cache. Three incremental directions extend the architecture: selective inlining for hot operators, operator-level glue-code caching, and glue-code compaction for wide segments.

8 Related Work

Compilation-based query execution. Compiling queries into executable code dates back to System R [7]. Gamma [9] compiled

predicates into machine language, HIQUE [19] generated query-specific C code, and HyPer [23] advanced this with LLVM IR. Spark SQL [1] applied code generation for distributed processing. All generate operator *logic* at query time. Composition eliminates this cost. Caching and serverless compilation [8] reduce but do not eliminate the overhead, and composition coexists with both.

Vectorized execution. MonetDB/X100 [6] amortizes interpretation overhead by processing batches through pre-compiled primitives. VIP [28] builds an engine entirely from pre-compiled SIMD sub-operators. Kersten et al. [15] showed that neither compilation nor vectorization dominates across query shapes. Photon, Velox, and DuckDB chose vectorization specifically to avoid the compilation cold-start: Photon [3] cites engineering simplicity and predictable latency for Databricks' Lakehouse; Velox [27] provides a shared vectorized engine across Meta's data platforms; DuckDB [29] forgoes query-time compilation entirely. The trade-off in all three is that batch materialization at operator boundaries forfeits the cross-operator fusion that compilation provides. Composition takes the opposite position: it preserves the compiled execution model (no batched intermediates) while using pre-compiled operators to bound query-time cost like vectorization.

Relaxed operator fusion. Menon et al. [20] insert staging points within compiled pipelines for prefetching. ROF modifies fusion *granularity*, while composition modifies the *cost* of producing the executable. The two are complementary.

Composable kernels and tracing-based engines. BOSS [22] composes independent kernels for hardware portability but does not address compilation cost. Nautilus [12] uses a trace-based JIT that still specializes operators on the critical path. FastCompose adds a composition path with no query-time specialization.

Adaptive execution. Umbra [4, 16, 24] uses adaptive execution with a custom IR and two compilation backends (a fast direct-to-x86 backend [16] and LLVM). Both still generate operator *logic* at query time, reducing but not eliminating compilation cost. ReSQL [10] takes a similar low-latency angle by designing Flounder IR, a lightweight IR close to machine assembly that is faster to translate than LLVM IR; it still performs the translation at query time. VOILA [13] parameterizes the execution strategy per operator. ClickHouse [30] adaptively JITs expressions and aggregations after repeated execution. PCQ [21] inserts indirection for runtime adaptivity without recompilation. Kohn et al. [18] switch between interpretation and compiled execution as the compiler catches up. All generate operator logic at query time. Composition differs fundamentally: operators are already in the binary, and only glue code is generated, bounding cost to operator count rather than code size. Umbra also requires a custom compiler backend [16], whereas composition reuses LLVM.

Staged code generation. LegoBase [17] and LB2 [31] use multi-stage programming to generate efficient query code. Weld [25] provides a common IR for cross-library optimization. FastCompose's injector serves an analogous role but operates on *existing* C++ operator implementations rather than requiring a new DSL.

Copy-and-patch compilation. Xu and Kjolstad [34] introduce copy-and-patch, a technique that pre-compiles code "stencils" at build time and stitches them into native binary code at runtime, achieving compilation speeds two orders of magnitude faster than LLVM. Applied to database query compilation, copy-and-patch operates at the instruction level (each stencil is a small code fragment), whereas composition operates at the operator level, wiring pre-compiled operator implementations through glue code. Composition also integrates with an existing compilation path (the Injector generates both outputs from the same operator source), while copy-and-patch replaces the compiler entirely.

Threaded code. Composition shares roots with subroutine threading [5], but differs in three ways: it generates query-specific native code with full control flow rather than dispatching through a table, it serves as a temporary bridge replaced by fully optimized compiled code, and the injector enables both modes from shared operator code.

Partial evaluation and specialization. GraalVM's Truffle [33] shares composition's "pre-compile then arrange" philosophy but specializes interpreter nodes through profile feedback, while composition arranges operators through query-plan-driven glue code without profiling.

9 Conclusion

The cold-start problem in compilation-based query execution is not inherent. Composed code executes the same fully optimized operators as compiled code, losing only cross-boundary optimization. This observation enabled us to build FastCompose, which separates arrangement from logic: operators are compiled once at build time, and only lightweight glue code is generated per query. Composed code executes immediately while compilation proceeds in the background, and the compiled code cache ensures that subsequent executions of the same query run at full compiled performance.

In production across thousands of Amazon Redshift clusters, FastCompose reduces median per-segment compilation time by 61× and cold-start dashboard latency by 3.5×, with composed execution within 2–10% of fully compiled performance. The same approach extends to expressions: pre-compiled scalar functions are called through glue code with strategic inlining to bound composition time without sacrificing per-tuple performance. Two directions for future work remain: glue code compaction for wide tables and deep joins, and reducing the directive burden through automatic type inference. More broadly, composition demonstrates that pre-compiled operators can be arranged into native executables without cross-boundary compilation, suggesting new trade-offs in execution engine design. Combining composition and compilation in a single engine yields both low cold-start latency and peak steady-state throughput.

Acknowledgments

We thank David DeWitt for his detailed review and feedback on early drafts of this paper, and Ippokratis Pandis for his involvement in the design of FastCompose and the entire Amazon Redshift team for their support. We are grateful to Sudipto Das, Nikos Armenatzoglou, and Yanlei Diao for their feedback and reviews on this paper, and to Andy Warfield, Mai-Lan Tomsen Bukovec, and James Hamilton for their support and encouragement. We acknowledge Manuel Lux, Ioanna Tsalouchidou, Anton Kovalev, Robert Brunel, Yiqing Wang, Ayush Kumar, Hui Shi, Adekunle Adedotun, Srinath Madabushi, Amol Mhatre, Hanna Loboda, Stefan Gromoll, Indu Bhagavatula, and Ravi Animi for their contributions to the development and deployment of FastCompose, and the Redshift customers whose workloads and feedback motivated this work.

References

[1] Michael Armbrust, Reynold S. Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K. Bradley, Xiangrui Meng, Tomer Kaftan, Michael J. Franklin, Ali Ghodsi, and Matei Zaharia. 2015. Spark SQL: Relational Data Processing in Spark. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data* (Melbourne, Victoria, Australia) *(SIGMOD '15)*. Association for Computing Machinery, New York, NY, USA, 1383–1394. doi:10.1145/2723372.2742797

[2] Nikos Armenatzoglou, Sanuj Basu, Naga Bhanoori, Mengchu Cai, Naresh Chainani, Kiran Chinta, Venkatraman Govindaraju, Todd J. Green, Monish Gupta, Sebastian Hillig, Eric Hotinger, Yan Leshinksy, Jintian Liang, Michael McCreedy, Fabian Nagel, Ippokratis Pandis, Panos Parchas, Rahul Pathak, Orestis Polychroniou, Foyzur Rahman, Gaurav Saxena, Gokul Soundararajan, Sriram Subramanian, and Doug Terry. 2022. Amazon Redshift Re-invented. In *Proceedings of the 2022 International Conference on Management of Data* (Philadelphia, PA, USA) *(SIGMOD '22)*. Association for Computing Machinery, New York, NY, USA, 2205–2217. doi:10.1145/3514221.3526045

[3] Alexander Behm, Shoumik Palkar, Utkarsh Agarwal, Timothy Armstrong, David Cashman, Ankur Dave, Todd Greenstein, Shant Hovsepian, Ryan Johnson, Arvind Sai Krishnan, Paul Leventis, Ala Luszczak, Prashanth Menon, Mostafa Mokhtar, Gene Pang, Sameer Paranjpye, Greg Rahn, Bart Samwel, Tom van Bussel, Herman van Hovell, Maryann Xue, Reynold Xin, and Matei Zaharia. 2022. Photon: A Fast Query Engine for Lakehouse Systems. In *Proceedings of the 2022 International Conference on Management of Data* (Philadelphia, PA, USA) *(SIGMOD '22)*. Association for Computing Machinery, New York, NY, USA, 2326–2339. doi:10.1145/3514221.3526054

[4] Alexander Beischl, Timo Kersten, Maximilian Bandle, Jana Giceva, and Thomas Neumann. 2021. Profiling dataflow systems on multiple abstraction levels. In *Proceedings of the Sixteenth European Conference on Computer Systems* (Online Event, United Kingdom) *(EuroSys '21)*. Association for Computing Machinery, New York, NY, USA, 474–489. doi:10.1145/3447786.3456254

[5] James R. Bell. 1973. Threaded code. *Commun. ACM* 16, 6 (June 1973), 370–372. doi:10.1145/362248.362270

[6] Peter A Boncz, Marcin Zukowski, and Niels Nes. 2005. MonetDB/X100: Hyper-Pipelining Query Execution.. In *Cidr*, Vol. 5. 225–237.

[7] Donald D. Chamberlin, Morton M. Astrahan, Michael W. Blasgen, James N. Gray, W. Frank King, Bruce G. Lindsay, Raymond Lorie, James W. Mehl, Thomas G. Price, Franco Putzolu, Patricia Griffiths Selinger, Mario Schkolnick, Donald R. Slutz, Irving L. Traiger, Bradford W. Wade, and Robert A. Yost. 1981. A history and evaluation of System R. *Commun. ACM* 24, 10 (Oct. 1981), 632–646. doi:10.1145/358769.358784

[8] Kiran Chinta, Quan Li, Maor Kleider, and Naresh Chainani. 2020. Fast and predictable performance with serverless compilation using Amazon Redshift. AWS Big Data Blog. https://aws.amazon.com/blogs/big-data/fast-and-predictable-performance-with-serverless-compilation-using-amazon-redshift/.

[9] D. J. Dewitt, S. Ghandeharizadeh, D. A. Schneider, A. Bricker, H. I. Hsiao, and R. Rasmussen. 1990. The Gamma Database Machine Project. *IEEE Trans. on Knowl. and Data Eng.* 2, 1 (March 1990), 44–62. doi:10.1109/69.50905

[10] Henning Funke, Jan Mühlig, and Jens Teubner. 2022. Low-latency query compilation. *The VLDB Journal* 31, 6 (May 2022), 1171–1184. doi:10.1007/s00778-022-00741-5

[11] G. Graefe. 1994. Volcano — An Extensible and Parallel Query Evaluation System. *IEEE Trans. on Knowl. and Data Eng.* 6, 1 (Feb. 1994), 120–135. doi:10.1109/69.273032

[12] Philipp M. Grulich, Aljoscha P. Lepping, Dwi P. A. Nugroho, Varun Pandey, Bonaventura Del Monte, Steffen Zeuch, and Volker Markl. 2024. Query Compilation Without Regrets. *Proc. ACM Manag. Data* 2, 3, Article 165 (May 2024), 28 pages. doi:10.1145/3654968

[13] Tim Gubner and Peter Boncz. 2021. Charting the design space of query execution using VOILA. *Proc. VLDB Endow.* 14, 6 (Feb. 2021), 1067–1079. doi:10.14778/3447689.3447709

[14] Anurag Gupta, Deepak Agarwal, Derek Tan, Jakub Kulesza, Rahul Pathak, Stefano Stefani, and Vidhya Srinivasan. 2015. Amazon Redshift and the Case for Simpler Data Warehouses. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data* (Melbourne, Victoria, Australia) *(SIGMOD '15)*. Association for Computing Machinery, New York, NY, USA, 1917–1923. doi:10.1145/2723372.2742795

[15] Timo Kersten, Viktor Leis, Alfons Kemper, Thomas Neumann, Andrew Pavlo, and Peter Boncz. 2018. Everything you always wanted to know about compiled and vectorized queries but were afraid to ask. *Proc. VLDB Endow.* 11, 13 (Sept. 2018), 2209–2222. doi:10.14778/3275366.3284966

[16] Timo Kersten, Viktor Leis, and Thomas Neumann. 2021. Tidy Tuples and Flying Start: fast compilation and fast execution of relational queries in Umbra. *The VLDB Journal* 30, 5 (2021), 883–905. doi:10.1007/s00778-020-00643-4

[17] Yannis Klonatos, Christoph Koch, Tiark Rompf, and Hassan Chafi. 2014. Building efficient query engines in a high-level language. *Proc. VLDB Endow.* 7, 10 (June 2014), 853–864. doi:10.14778/2732951.2732959

[18] André Kohn, Viktor Leis, and Thomas Neumann. 2018. Adaptive Execution of Compiled Queries. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*. 197–208. doi:10.1109/ICDE.2018.00027

[19] Konstantinos Krikellas, Stratis D. Viglas, and Marcelo Cintra. 2010. Generating code for holistic query evaluation. In *2010 IEEE 26th International Conference on Data Engineering (ICDE 2010)*. 613–624. doi:10.1109/ICDE.2010.5447892

[20] Prashanth Menon, Todd C. Mowry, and Andrew Pavlo. 2017. Relaxed operator fusion for in-memory databases: making compilation, vectorization, and prefetching work together at last. *Proc. VLDB Endow.* 11, 1 (Sept. 2017), 1–13. doi:10.14778/3151113.3151114

[21] Prashanth Menon, Amadou Ngom, Lin Ma, Todd C. Mowry, and Andrew Pavlo. 2020. Permutable compiled queries: dynamically adapting compiled queries without recompiling. *Proc. VLDB Endow.* 14, 2 (Oct. 2020), 101–113. doi:10.14778/3425879.3425882

[22] Hubert Mohr-Daurat, Xuan Sun, and Holger Pirk. 2025. BOSS - An Architecture for Database Kernel Composition. *SIGMOD Rec.* 54, 1 (April 2025), 37–46. doi:10.1145/3733620.3733629

[23] Thomas Neumann. 2011. Efficiently compiling efficient query plans for modern hardware. *Proc. VLDB Endow.* 4, 9 (June 2011), 539–550. doi:10.14778/2002938.2002940

[24] Thomas Neumann and Michael J. Freitag. 2020. Umbra: A Disk-Based System with In-Memory Performance. In *Conference on Innovative Data Systems Research*. https://api.semanticscholar.org/CorpusID:209379505

[25] Shoumik Palkar, James J Thomas, Anil Shanbhag, Deepak Narayanan, Holger Pirk, Malte Schwarzkopf, Saman Amarasinghe, and Matei Zaharia. 2017. Weld: A Common Runtime for High Performance Data Analytics. In *CIDR*.

[26] Ippokratis Pandis, Naresh Chainani, Kiran Kumar Chinta, Venkatraman Govindaraju, Andrew Edward Caldwell, Naveen Muralimanohar, Martin Grund, Fabian Oliver Nagel, and Nikolaos Armenatzoglou. 2023. Compiled Query Plan Cache and Reuse. US Patent 11,853,301.

[27] Pedro Pedreira, Orri Erling, Masha Basmanova, Kevin Wilfong, Laith Sakka, Krishna Pai, Wei He, and Biswapesh Chattopadhyay. 2022. Velox: meta's unified execution engine. *Proc. VLDB Endow.* 15, 12 (Aug. 2022), 3372–3384. doi:10.14778/3554821.3554829

[28] Orestis Polychroniou and Kenneth A. Ross. 2020. VIP: A SIMD vectorized analytical query engine. *The VLDB Journal* 29, 6 (July 2020), 1243–1261. doi:10.1007/s00778-020-00621-w

[29] Mark Raasveldt and Hannes Mühleisen. 2019. DuckDB: an Embeddable Analytical Database. In *Proceedings of the 2019 International Conference on Management of Data* (Amsterdam, Netherlands) *(SIGMOD '19)*. Association for Computing Machinery, New York, NY, USA, 1981–1984. doi:10.1145/3299869.3320212

[30] Robert Schulze, Tom Schreiber, Ilya Yatsishin, Ryadh Dahimene, and Alexey Milovidov. 2024. ClickHouse - Lightning Fast Analytics for Everyone. *Proc. VLDB Endow.* 17, 12 (Aug. 2024), 3731–3744. doi:10.14778/3685800.3685802

[31] Ruby Y. Tahboub, Grégory M. Essertel, and Tiark Rompf. 2018. How to Architect a Query Compiler, Revisited. In *Proceedings of the 2018 International Conference on Management of Data* (Houston, TX, USA) *(SIGMOD '18)*. Association for Computing Machinery, New York, NY, USA, 307–322. doi:10.1145/3183713.3196893

[32] Alexander van Renen, Dominik Horn, Pascal Pfeil, Kapil Vaidya, Wenjian Dong, Murali Narayanaswamy, Zhengchun Liu, Gaurav Saxena, Andreas Kipf, and Tim Kraska. 2024. Why TPC is Not Enough: An Analysis of the Amazon Redshift Fleet. *Proc. VLDB Endow.* 17, 11 (July 2024), 3694–3706. doi:10.14778/3681954.3682031

[33] Thomas Würthinger, Christian Wimmer, Christian Humer, Andreas Wöß, Lukas Stadler, Chris Seaton, Gilles Duboscq, Doug Simon, and Matthias Grimmer. 2017. Practical partial evaluation for high-performance dynamic language runtimes. *SIGPLAN Not.* 52, 6 (June 2017), 662–676. doi:10.1145/3140587.3062381

[34] Haoran Xu and Fredrik Kjolstad. 2021. Copy-and-patch compilation: a fast compilation algorithm for high-level languages and bytecode. *Proc. ACM Program. Lang.* 5, OOPSLA, Article 136 (Oct. 2021), 30 pages. doi:10.1145/3485513