Rhyme Native: Efficient Code Generation for Structured and Semi-Structured Workloads

Ran Guo
Department of Computer Science
Purdue University
West Lafayette, Indiana, USA
guo543@purdue.edu

Tiark Rompf
Department of Computer Science
Purdue University
West Lafayette, Indiana, USA
tiark@purdue.edu

ABSTRACT

Modern data processing spans two worlds: flat relational tables, served by decades of database research producing highly optimized query engines, and nested semi-structured data such as JSON, for which expressive query languages exist but compilation and optimization techniques have been applied far less comprehensively. We ask whether a single query language can express both regimes naturally while compiling to efficient native code. We build on Rhyme, a declarative language whose object-notation syntax mirrors the structure of query results, and contribute on three fronts. We refine Rhyme's semantics for generator binding and missing values, allowing co-iteration, inner/outer joins, and nested-loop traversals to be expressed under different uses of generator symbols. We show that Rhyme's prior dependency-driven loop scheduler can generate incorrect code on hierarchical queries, and present a new scheduler based on finer-grained per-statement constraints that ensures correctness. We introduce a gradual type system and a C code generation backend that emits tag-less, statically typed code and specializes data loading and internal data structures for idiomatic SQL patterns. Our system matches state-of-the-art compiled engines on SQL workloads such as TPC-H and outperforms modern JSON-capable databases and DSLs on JSONBench and other hierarchical queries.

PVLDB Reference Format:
Ran Guo and Tiark Rompf. Rhyme Native: Efficient Code Generation for Structured and Semi-Structured Workloads. PVLDB, 19(11): 3676 - 3688, 2026.
doi:10.14778/3836663.3836717

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/rhyme-lang/rhyme.

1 INTRODUCTION

Modern data processing workloads can be broadly categorized into two distinct types based on their underlying data structures and query patterns. The first category encompasses SQL-style queries over traditional relational data with flat, tabular structures where data is organized in rows and columns with well-defined schemas. The second category involves queries over nested, hierarchical data such as JSON objects, XML documents, and other tree-like representations that are increasingly prevalent in web applications, APIs, and document databases.

DB Systems vs DSLs. Mature database systems excel in the first category. Traditional systems such as PostgreSQL [5] and modern analytical databases like DuckDB [35] provide robust support for relational operations including joins, aggregations, and complex analytical queries. In-memory databases like HyPer [25, 31] and its successor Umbra [32] have pushed performance further by incorporating native code generation. Systems like Flare [21], built on top of Spark [12], also aim to compile SQL queries into high-performance C code with the help of LMS [39] via staging. Compilation techniques [14, 27, 31, 42] have further pushed the boundaries of performance for relational workloads. However, these systems struggle with the second category of workloads. While modern SQL systems can load and query JSON data, SQL lacks the natural expressiveness needed for hierarchical operations with nested results, often forcing users into less ergonomic nested subqueries.

Specialized domain-specific languages (DSLs) have emerged to address nested data querying needs. Declarative languages such as JSONiq [3, 37], JQ [2], and JSONPath [22] provide powerful primitives for JSON manipulation, while XQuery [7] and GraphQL [1] offer declarative query capabilities for XML and nested API responses, respectively. However, they sit at a different stage in terms of optimizations compared with relational queries. Relational data processing has had decades of optimization—cost-based optimization, vectorized execution, native code generation—leaving increasingly little room for further gains. These DSLs, by contrast, are designed for convenient processing of nested data and are most often interpreted and have not had compilation techniques applied as comprehensively. As a result, they lack the performance and broad applicability needed for workloads spanning both structured and semi-structured data—a gap that is also an opportunity, leaving substantial headroom for such systems.

The Need for Elaborate Queries. Meaningful analysis over semi-structured data goes beyond basic path navigation and group-bys: users need co-iterations across collections on shared keys (inner joins), nested-loop products of two collections, outer joins, and similar combinators. Such operations are difficult to express in stream-based languages like JQ, which process a single sequence through a pipeline and provide no first-class way to iterate two collections jointly; joins and cross-products must be hand-coded.

Systems like Proteus [24] address this with an extended relational algebra over heterogeneous data, specializing the engine via LLVM code generation. The programming model, however, remains

Proceedings of the VLDB Endowment, Vol. 19, No. 11 ISSN 2150-8097.
doi:10.14778/3836663.3836717

fundamentally relational—queries are compositions of algebraic operators that consume nested data but are not shaped by it. Producing hierarchical results, multi-level group-bys, or correlated nested aggregations still requires encoding tree-shaped intent through flat relational operations.

The ultimate goal is a unified query language and compiler in which queries are both natural to write and efficient to execute, regardless of whether the underlying data is flat or deeply nested. This goal, however, remains challenging, and the main challenges can be categorized into three aspects: the user-facing query language level, the logical level, and the physical level.

Expressive Query Language. The first challenge is a language with clean syntax for path access, aggregations, nested-result construction, and controllable iteration—co-iteration vs. nested loop—over collections at different nesting levels. **Rhyme** [9, 10] was proposed as an expressive query language designed for nested, hierarchical data. Queries in Rhyme are themselves structured *objects* represented using object notation, where the query structure directly corresponds to the result structure, enabling natural authoring of hierarchical queries and iterations are expressed implicitly via generator symbols. The natural and concise syntax of Rhyme allows users to express diverse styles of computations including traditional relational queries on tabular data along with hierarchical, semi-structured data in a unified language. However, the binding semantics of generators across different prefixes, outer joins, and the handling of absent values were not clearly established in prior work. We refine these semantics (Section 2), letting generator symbols express co-iteration, nested loops, inner joins, and outer joins as concise idioms over collections.

Logical Layer: Intermediate Representation. The second challenge involves the design of the intermediate representation (IR) for hierarchical queries. In traditional relational engines, the execution of queries follows the construction of a query plan—a tree of operators whose order is committed by the optimizer which often performs cost-based optimizations (e.g., selectivity estimation). Tuples in tables then flow through a pipeline of relational operators, classically realized via the Volcano iterator model. Modern compiled engines such as HyPer [31] and Umbra [32] go a step further and use data-centric code generation to fuse operator pipelines into tight loops, eliminating per-tuple dispatch. However, the loop shape generated from the plan is predictable after the plan is established.

Code generation for hierarchical queries is fundamentally different. A Rhyme query is a single expression whose result may itself be a multi-level nested object; iterations, aggregations, joins, and result construction are intermixed inside one declarative term. The challenge lies in identifying the correct and optimal looping structure—whether loops should be nested, hoisted, or fused, and whether an aggregation must be fully materialized before its value is read. A relational query plan does not capture any nested traversals or loop shaping constraints. Cost-based optimization also becomes much harder with tree-shaped data. There exists no analogous query plan in Rhyme. Rhyme's IR follows from a dependency analysis over the IR statements, and the loop scheduler synthesizes the final loop nest based on the dependency graph. The loop-free, branch-free IR together with the loop scheduler is the distinctive approach that drives code generation from Rhyme queries.

We identify a class of queries on which the prior dependency-driven scheduler [9, 10] generates incorrect code due to an overly coarse-grained loop-constraint analysis, and design a new scheduler based on fine-grained per-statement constraints and proper topological sort that ensures correctness (Section 3). We show that the new loop scheduling strategy solves issues in the prior scheduler. While the present loop scheduler does not perform cost-based analysis (we consider this beyond the scope of this paper), we discuss how it could be extended in this way.

Physical Layer: Native Codegen. Native code generation for hierarchical queries has received less attention than for relational queries. Rhyme's existing JS backend is limited by JavaScript's dynamic nature, and many Rhyme queries fall into idiomatic SQL patterns over flat tables that benefit from specialized native compilation. In this work, we extend Rhyme with a C code generator.

Traditional database systems operate on tabular data with fixed schema. In hierarchical queries, by contrast, the intermediate state varies per query rather than being fixed by any schema—scalars, arrays, hash tables, or hash tables whose values are themselves hash tables can all arise depending on the query's syntactic shape. To support efficient native code generation over dynamic JSON data and to enable correct specialization of these intermediate data structures, we add a type system to Rhyme so that users can optionally provide an input schema. With the type information, the C code generation emits tag-less code over concretely typed values and selects appropriate specialized internal data structures for each value shape. We further accelerate the SQL-style subset of Rhyme queries by recognizing idiomatic SQL patterns and specializing the data loading and internal data structures in the generated code (Section 4). The C code generation backend delivers substantial performance improvements for queries over semi-structured data while matching the performance of state-of-the-art compiled query engines on traditional SQL workloads.

Our main contributions are as follows:

- We refine Rhyme's semantics on generator binding, unlocking concise expression of co-iteration, nested-loop traversals, and inner/outer joins (Section 2).
- We present an improved dependency-driven scheduling strategy for synthesizing loop structures from loop-free IR (Section 3).
- We equip Rhyme with a gradual type system that enables efficient code generation and catches ill-typed queries statically (Sections 2 and 4).
- We describe a C code generation backend with specialized optimizations for SQL-style queries over tabular data, including pattern-based recognition of relational idioms and efficient internal data structures to store results (Section 4).
- We evaluate the system on TPC-H, JSONBench, and additional hierarchical queries, demonstrating performance matching Flare and CedarDB on TPC-H while maintaining expressive power beyond SQL (Section 5).

2 THE RHYME QUERY LANGUAGE

This section presents the Rhyme query language, providing essential background for understanding the IR and loop scheduling. We first introduce Rhyme's syntax and core operations through examples. We then describe the refined semantics for handling missing

values and generator symbol binding. Finally, we present the type system that enables statically typed code generation.

We introduce Rhyme's core operations through examples using the following input data:

```
let data = [
  { country: "Japan",  city: "Tokyo",      population: 14 },
  { country: "China",  city: "Beijing",    population: 22 },
  { country: "UK",     city: "London",     population: 9 },
  { country: "Japan",  city: "Osaka",      population: 3 },
  { country: "UK",     city: "Birmingham", population: 2 }
]
```

Path Expressions. Rhyme uses path expressions (inspired by JSONPath [22]) to access nested values. The dot operator (`.`) accesses fields, and arrays are indexed with integers; e.g., `data.0.city` evaluates to `"Tokyo"`.

Iterations & Aggregations. Inspired by generators in JQ [2], Rhyme uses generator symbols (`.*`) in path expressions to express iterations, where the prefix specifies the collection to iterate.

```
sum   data.*.population   // result: 50
count data.*              // result: 5
[data.*.city]             // result: ["Tokyo", "Beijing", ...]
```

Nested Query Structures, Nested Results. Rhyme queries use object literal notation to construct structured output, where the query structure directly corresponds to the result structure:

```
{ third: data.2.country } // result: { third: "UK" }
```

Group-by operations can be intuitively expressed as JSON objects where keys define the grouped key and values define aggregations.

```
{ data.*.country: sum data.*.population }
// result: {Japan: 17, China: 22, UK: 11}
```

The aggregation computes the sum for each unique value of `country`. The object key contains the same iteration symbol `*`, establishing correlation between the aggregation and group key—the sum is scoped to each country. Iterations nested under this key execute for each unique country value. Multi-level group-bys and aggregations compose naturally, with the object structure directly reflecting the result structure:

```
{                                    // {
  total: sum data.*.population,      //   total: 50,
  data.*.country: {                  //   Japan: {
    total: sum data.*.population,    //     total: 17,
    data.*.city:                     //     Tokyo: 14,
      sum data.*.population          //     Osaka: 3
  }                                  //   }
}                                    // }
```

The sum at each nesting level computes distinct results: total population, per-country totals, and per-city populations. The corresponding result is shown on the right.

Missing Values & Filtering. The handling of missing values has long been a source of ambiguity in SQL: different systems disagree on how nulls behave under various operations, and the topic remains an active subject of research [17, 23, 36, 40, 46].

Similarly, in JavaScript, a failed property access yields `undefined`. This can arise in Rhyme queries from dynamic lookups or from malformed records (e.g., some objects carry a field that others lack). Stateful operations—aggregations and group-bys—uniformly ignore `undefined` values rather than propagating them.

This mechanism doubles as a filter: producing an `undefined` value at a position is the same as omitting that position from an aggregation. Comparison operators are designed accordingly—they produce either `true` or `undefined` rather than `true`/`false`. Combined with the short-circuiting `and` operator (`&`)—which returns the first falsy value or the right-hand side if none—this gives a concise filtering idiom:

```
[data.*.population > 10 & data.*.city]
// result: ["Tokyo", "Beijing"]
```

When the predicate evaluates to `undefined`, the entire expression will evaluate to `undefined` and the aggregation ignores that value, effectively filtering the cities with the given predicate.

Anonymous vs. Named Generators. Generators may also be *named* (e.g., `*A`, `*B`, ...) by appending an identifier after the asterisk. In prior work [9, 10], anonymous `*` after different prefixes denoted a shared co-iteration, so a nested-loop join had to be expressed by explicitly using distinct named generators (e.g., `a.*A` and `b.*B`); only inner joins were available, with no concise way to express outer joins. We refine the binding rule of anonymous and named generators so that anonymous `*` after different prefixes denotes *distinct* generators (a nested loop) while shared named generators `*A` denote co-iteration on the intersection of keys, and we add a `?` suffix for missing-key tolerance. Together these changes unlock co-iteration, nested-loop iterations and inner/outer joins.

Concretely, each occurrence of an anonymous `*` is bound to its prefix and is therefore a fresh generator: `data.*` and `other.*` are two independent iterations, canonicalized internally to distinct symbols (`*1`, `*2`). Named generators behave the opposite way: `data.*A` and `other.*A` refer to the *same* generator `*A` shared across both prefixes, behaving like logic variables in Datalog [16] and Prolog [18]. Consider the data:

```
let a = [1, 2, 3, 4]      let c = { A: 1, B: 2, C: 3 }
let b = [5, 6, 7]         let d = { B: 2, C: 3, D: 4 }
```

A shared named generator co-iterates the collections over their key intersection: for `a.*A` and `b.*A`, the loop ranges over keys of `a` and skips any absent from `b`. Since JavaScript array keys are integer indices, co-iterating arrays is a zip; for objects it is an inner join on shared keys:

```
sum (a.*A * b.*A)         [c.*A + d.*A]
// result: 38             // result: [4, 6]
// (dot product)          // (Only taking sum of B's and C's)
```

Appending `?` after the generator skips the existence check on its prefix, recovering outer-join semantics; missing values must be handled explicitly:

```
sum (a.*A + (b.*A? || 1))
// result: 29 (defaults to 1 on missing values in b)
```

By contrast, distinct anonymous generators yield a nested-loop iteration, since `a.*` and `b.*` canonicalize to different symbols:

```
[{ fst: a.*, snd: b.* }]
// result: [{ fst: 1, snd: 5 }, { fst: 1, snd: 6 }, ...]
```

These idioms—inner join, outer join, and nested-loop product—have no equally concise counterpart in stream-based JSON languages. JQ [2] processes a single stream of values and offers no way to name an iteration shared across two collections, so even the simplest co-iteration must enumerate the keys of one collection, test membership in the other, and index both explicitly. The dot product `sum (a.*A * b.*A)` above must be written as:

```
[$a | keys_unsorted[] as $k | select($b | has($k))
    | $a[$k] * $b[$k]] | add
```

Named Generators as Values. Because named generators are shared by name, they are first-class identifiers and can be used freely in any expression. A bare `*A` refers to the current array index or object key, so the key can participate directly in computations or as group keys. For example, the following pairs each record with its position in `data` under a single `*A` iteration:

```
[{p: data.*A.population, i: *A + 1}]
// result: [{p: 14, i: 1}, {p: 22, i: 2}, {p: 9, i: 3}, ...]
```

Piped Operations. While object literal syntax reflects result structure, transformation-oriented queries become unwieldy with deeply nested calls. Rhyme's pipe operator (`|`) enables intuitive sequential transformations. E.g., finding the largest per-country population in the running dataset reads as a transformation chain:

```
.data.*A.population | sum | group data.*A.country | .* | max
```

Loading Input from Files. Rhyme provides `load<file type>` operations for runtime file loading:

```
sum (loadJSON "data.json").*.population
```

CSV and TBL files are supported via `loadCSV` and `loadTBL`; both expose rows as JSON arrays of column-keyed objects, so they can be queried with the same operators as any other input. This unified data model lets a single query mix structured and semi-structured sources elegantly in a single language.

Typing. To facilitate efficient C code generation, we have built a type system which ensures query type safety and early error detection. Some aggregation operations require specific types (e.g., `sum` requires numeric types). Users can specify optional input data schemas via provided APIs. When schemas are provided, the compiler performs type checking to ensure query type safety. Additionally, type information enables native code generation to eliminate tagged unions in generated C code, avoiding runtime overhead. Rhyme supports gradual typing: schemas are optional and can be partial, and absent types are often inferred from context (e.g., `sum` forces a numeric result), so a query compiles and runs even with partial or fully unknown types. A schema is written in the same object notation as queries and results—an object whose structure mirrors the nested shape of the input data. Type inference annotates every subexpression with a type—a primitive type for scalar results, or an object type for collections—that the C backend consumes directly to select the appropriate internal data structure (Section 4).

Rhyme supports the usual primitive fixed-width integer and floating-point types, plus a base `string` type and constant strings, chars, etc. Schemas are passed alongside the Rhyme queries to the compiler.

Object Types. Object schemas reuse the same object literal notation as queries and results. Dynamic object keys with known type require a key type, denoted by prefixing `*` before the type. Rhyme unifies objects and arrays, treating arrays as objects with integer keys. For example, an array of objects with fields `k` (string) and `v` (integer) is specified as:

```
{ *u32: { k: string, v: u32 } }
```

Table 1: Source queries and extracted aggregations.

Query	Source & Extracted Aggregations

(a)	`sum data.*.population`
	`----- Aggregations -----`
	`sum data.*.population`
	`// group path: []`

(b)	`{ data.*.country: sum data.*.population }`
	`----- Aggregations -----`
	`sum data.*.population`
	`// group path: [data.*.country]`

(c)	`{ data.*A.country: sum data.*A.population /`
	`sum data.*B.population }`
	`----- Aggregations -----`
	`sum data.*A.population`
	`// group path: [data.*A.country]`
	`sum data.*B.population`
	`// group path: []`

3 IR AND LOOP SCHEDULING

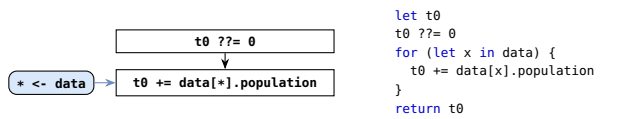
A Rhyme query carries no explicit loops, so synthesizing an efficient and correct loop nest from it is entirely the compiler's responsibility. This section presents the loop-free IR that records only inter-statement dependencies and the scheduler that derives the loop structure from them. The scheduler is the crux: we exhibit a class of queries on which Rhyme's prior scheduler silently emits incorrect loops (sibling aggregations writing into a shared nested result) and replace its coarse per-temporary loop constraints with per-statement constraints that restore correctness.

3.1 Rhyme IR

This subsection introduces Rhyme's intermediate representation (IR) structure and its construction from input queries. We present examples of IR dependencies and the resulting generated code.

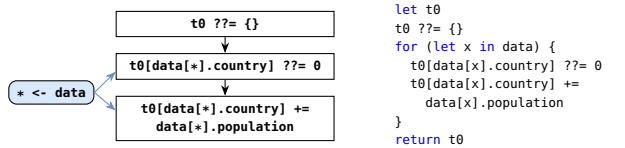
Generator and Assignment Statements. Rhyme's IR consists of two statement types: *generators* represent iterations over iterable data (e.g., JS arrays or objects), while *assignments* represent computations that produce results. Generators turn into loops in generated code; assignments update temporary state variables for *stateful* operations (e.g., aggregations/group-bys). Crucially, the IR encodes dependencies between statements rather than explicit loop structures. The code structure is derived from these dependencies by the loop scheduler.

Table 1 illustrates the extraction of aggregations from source queries and Figure 1 shows the corresponding IR statements (generators in colored round rectangles and assignments in rectangles) and the final generated code for each example in Table 1. Each



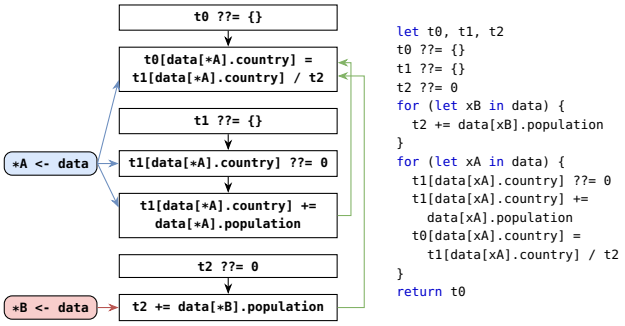

```
let t0
t0 ??= 0
for (let x in data) {
  t0 += data[x].population
}
return t0
```

(a) Sum of population




```
let t0
t0 ??= {}
for (let x in data) {
  t0[data[x].country] ??= 0
  t0[data[x].country] +=
    data[x].population
}
return t0
```

(b) Sum of population grouped by country




```
let t0, t1, t2
t0 ??= {}
t1 ??= {}
t2 ??= 0
for (let xB in data) {
  t2 += data[xB].population
}
for (let xA in data) {
  t1[data[xA].country] ??= 0
  t1[data[xA].country] +=
    data[xA].population
  t0[data[xA].country] =
    t1[data[xA].country] / t2
}
return t0
```

(c) Proportion of population for each country

Figure 1: IR and generated code of Rhyme queries. Arrows show dependencies between statements.

aggregation in the query AST generates multiple assignment statements, with path expressions translated to embedded JS expressions. Aggregations require state management through temporary variables. For example, row (a)'s sum operation creates temporary variable `t0`, initialized to 0, then updated by accumulating population values. The summed value transforms to the JS expression `data[x].population`.

IR Dependencies. In row (a) (Figure 1a), the `.*` path in `t0 += data[*].population` indicates a generator-to-assignment dependency on `* <- data`, and write ranks order the two writes to `t0` (initialization at rank 1, update at rank 2). The generated code emits the initialization outside the loop since it does not depend on `*`.

Correlated Aggregation with Group Path. For group-by operations, the grouped key expression indexes the temporary holding the aggregation result. Row (b) computes the sum of population per country; the sum's group path is `data.*.country` because its iteration `*` also appears in the group key. The three constructed statements—initializing `t0` to `{}`, initializing each `t0` entry indexed by `data[*].country` to 0 via `??=`, and updating with `+=`—all depend on the `* <- data` generator (Figure 1b), so the latter two are emitted inside the `*` loop.

Loop Hoisting via Dependency Analysis. Row (c) computes the per-country population proportion with respect to the total population: the group key `data.*A.country` drives `*A`, and the value is a division of two sums—one over `*A` (per-country, correlated with the group key) and one over `*B` (uncorrelated, global). Figure 1c uses three temporaries: `t1` for the per-country sum under `*A`, `t2` for the global sum, and `t0` for the final division. Because `t2` has no dependency on `*A`, the scheduler hoists its loop out to run before `*A` rather than nesting them. The assignment to `t0` depends on the final writes of both `t1` and `t2` (green arrows in Figure 1c), so it is emitted only after both aggregations complete. The examples above demonstrate the IR structure and dependency model of Rhyme.

3.2 Loop Scheduling

The loop scheduler synthesizes the executable program by deriving the looping structure from the IR's generators, assignments, and their dependencies. The algorithm is language-independent: IR statements already carry target-language code (JS or C), so the same logic serves both backends.

Coarse-grained analysis on temporary variables. The original Rhyme scheduler [9] derives loop structure from coarse-grained dependencies aggregated at the level of temporary state variables. This granularity carries an implicit assumption: that each temporary is produced at a single point in the computation, as the output of a relational operator is. Hierarchical queries break this assumption—a single result object may be populated by sibling subqueries that iterate different collections, so its writes belong under different loops. We have found this granularity to be insufficient: in the presence of independent writes to a shared temporary, it can produce loop nests that violate the IR's dependency semantics and yield incorrect results. The query below illustrates the problem:

```
{ total: sum [data.*A.value].*B, n: count data.*A }
```

The `total` field materializes an array of values under `*A` and sums it under a fresh generator `*B`, while the sibling `n` counts `data` under `*A`. The outer object becomes a single temporary `t0` that receives writes from both fields, and `t1` accumulates the array. Since one write to `t0` happens under `*A` and another under `*B`, scheduling at temporary granularity infers every `t0` write as depending on both, producing this incorrect program (simplified for clarity):

```
t0.total ??= 0; t0.n ??= 0; t1 ??= [];
for (let xA in data) {
  t1.push(data[xA].value)
  t0.n += 1
  for (let xB in t1)     // *B inside *A: t1 still partial!
    t0.total += t1[xB] }
```

The generated code is incorrect on two counts. First, `*B` is opened inside `*A` immediately after each `push`, so every iteration of `*B` ranges over a partially constructed array. Second, because `*B` runs once per outer iteration, partial sums are accumulated repeatedly, overcounting the result.

The root cause lies in the prior scheduler's `tmpAfterLoop` constraint, which prevents a statement from being emitted until a loop has closed whenever it reads a temporary whose producer's writes are nested inside that loop. The constraint is materialized per temporary: the scheduler collects the producer's loops and subtracts those in which the consumer's temporary itself resides, on the assumption that two writes co-located in the same loop need no closure between them. The subtraction thus operates at temporary granularity, not per write. When the consumer carries unrelated writes inside the producer's loop—as `t0` does here, since `n` forces one of its writes into `*A`—the subtraction cancels the very loop

whose closure the read of `t1` requires, and the read is scheduled inside the loop still building it.

Fine-grained analysis on statements. The defect stems from collapsing distinct writes onto a single per-temporary constraint. We redesign the scheduler around per-statement relations: dependencies and loop constraints are tracked at the granularity of individual IR statements rather than temporaries, so two writes to the same temporary may depend on different statements and be enclosed by different loops. The scheduler topologically sorts the statements and guarantees that each statement is emitted exactly once and only under the loops that it depends on.

Dependency Analysis. The analysis begins by building a dependency graph where each node represents an assignment or generator statement (Figure 1). After topologically sorting it, we compute two key relations per assignment statement: `stmtdeps` (statements that must be emitted before this one) and `loopdeps` (loops it transitively depends on). Because these are tracked per statement rather than per temporary, the update to `t0.total` retains only its dependency on `*B`, correctly separating it from `t0.n`.

We then compute the auxiliary `stmt2stmtByLoop` relation, which tracks loops that must be closed before a statement can be emitted. For instance, if statement `s2` depends on `s1` and `s1` depends on loop `l1`, then `s2` can only be emitted after loop `l1` closes with `s1` inside.

In the running example, the `push` that grows `t1` depends on `*A`, whereas the update to `t0.total` depends on `*B` and reads `t1`. The `stmt2stmtByLoop` relation therefore records that `*A` must close before the read of `t1` is emitted, so that the sum sees a fully materialized array.

Code Emission. We then move on to the actual code emission process. Code emission maintains three auxiliary data structures: `openedLoops` tracks currently active loops, `emittedStmts` tracks emitted statements (each statement is emitted exactly once under its dependent loops, enabling dependency-driven loop hoisting as in Figure 1c), and `closedLoopByStmt` maps statements to their closed dependent loops.

Algorithm 1 presents the code emission algorithm. Starting with all assignment statements, the algorithm emits statements in topological order as their dependencies become satisfied and required loops open. The recursive *EmitCode* function represents loop nesting through recursion depth.

DepsAvailableStmts filters statements whose dependencies are satisfied: a statement is available when all dependent statements are emitted and all loops specified by the `stmt2stmtByLoop` relation are closed. In the running example, the update to `t0.total` remains unavailable while `*A` is still open—even after the corresponding `t1.push` has been emitted—and only becomes available once `*A` closes and `t1` is fully constructed. However, available statements may still require unopened loops. *EmittableStmts* further filters available statements to those whose dependent loops are open (by looking up `openedLoops`). The algorithm emits each emittable statement and updates `emittedStmts`, iterating until no more statements become emittable.

Once the inner loop exhausts emittable statements, *DepsAvailableStmts* identifies statements awaiting loop openings. If no such statements exist, emission at the current nesting level is complete. Otherwise, *NextLoop* selects which loop to open next. Multiple loops may be eligible at this point, and the choice influences both

Algorithm 1 Code Generation

Require: $stmts$: set of assignment statements to be emitted
Ensure: Emitted code with proper loop nesting
Helper functions:
DepsAvailableStmts: returns statements whose dependencies have been emitted and whose awaited loops are closed
EmittableStmts: returns statements that can be emitted immediately
NextLoop: selects the next loop to open

```
1:  function EmitCode(stmts)
2:      while DepsAvailableStmts(stmts) ≠ ∅ do
3:          emittable ← EmittableStmts(stmts)
4:          stmts ← stmts \ emittable
5:          while emittable ≠ ∅ do
6:              for all s ∈ emittable do
7:                  Emit(s)
8:              end for
9:              emittable ← EmittableStmts(stmts)
10:             stmts ← stmts \ emittable
11:         end while
12:         emittableAfterLoop ← DepsAvailableStmts(stmts)
13:         if emittableAfterLoop = ∅ then
14:             return stmts
15:         end if
16:         loop ← NextLoop(emittableAfterLoop)
17:         OpenLoop(loop)
18:         stmtsInLoop ← {s ∈ stmts | s depends on loop}
19:         remainingStmts ← EmitCode(stmtsInLoop)      ▷ Recurse
20:         CloseLoop(loop)
21:         stmts ← (stmts \ stmtsInLoop) ∪ remainingStmts
22:     end while
23:     return stmts
24: end function
```

the shape of the generated code and its locality. We currently adopt a simple greedy heuristic: the scheduler opens the candidate loop that maximizes the number of statements that become immediately emittable upon opening. After emitting the loop header and updating `openedLoops`, the algorithm recursively invokes *EmitCode* on all statements dependent on the opened loop. Upon recursion return, `stmts` is updated by removing emitted statements and adding back those that could not be emitted. Closing the loop updates `closedLoopByStmt`.

Tracing this on the running example: initializations are emitted at the outer level (no loop dependencies). The scheduler opens `*A` and emits both `t1.push` and the increment of `t0.n` inside it, then closes `*A`. The write to `t0.total` now becomes available—its `stmt2stmtByLoop` constraint requires `*A` to be closed so that `t1` is fully materialized—so the scheduler opens `*B` and emits the sum inside it, producing the correct result. Replacing per-temporary loop constraints with per-statement constraints eliminates the fusion that arose from conflating unrelated writes to a shared temporary.

4 C CODE GENERATION

This section describes how Rhyme generates specialized, efficient C code for semi-structured hierarchical workloads while achieving competitive performance on SQL-style queries.

C code generation starts from the Rhyme AST and produces C-level assignment and generator statements, which are then passed

to the loop scheduler. Translating Rhyme queries into this IR raises several challenges: selecting efficient data structures for results and intermediates, avoiding redundant lookups and updates over those structures, and minimizing data-movement overhead. These concerns are hard to settle early because loop structure is finalized only after scheduling. We describe how the backend addresses these challenges and present optimizations that substantially improve SQL-style workloads. The C backend uses YYJSON [8], a high-performance JSON library written in ANSI C, to enable efficient JSON parsing and iteration.

4.1 Internal Data Structures

Storing intermediate query results naively as JSON-document structures would make the generated code inefficient, since results and intermediate values are read and written frequently. The C backend instead uses specialized internal data structures.

The choice of data structure follows directly from the inferred type: recall from Section 2 that type inference annotates every (sub)expression with its type. For a collection, this type already records the key type and value type, which the backend reads off to declare the corresponding data structure—a scalar type yields a temporary of the concrete C type, while object and array types yield the hash tables and arrays described below. When a value's type is not known statically, it is kept as a tagged union value array.

For simple aggregations producing a single scalar value, the result is stored in a temporary variable whose concrete C type is selected from the aggregation result type. Collections require more care. Array aggregations pair a C array with a companion length counter, and multi-component types use parallel arrays in a structure-of-arrays layout—a string array, for instance, becomes separate pointer and length arrays. Favoring such flat local arrays over nested structures improves register allocation and cache locality and exposes vectorization to the C compiler.

```
// Array of i32                 // Array of string
int32_t *t0 = malloc(...);      const char **t1_str = malloc(...);
int t0_count = 0;               int *t1_len = malloc(...);
                                int t1_count = 0;
```

Group-by operations require hash tables to store key-value mappings. The hash table comprises a main `htable` array storing indices into separate key and value arrays that hold the actual entries. Hash lookups compute a modulo hash to determine the key's position in an open-addressing `htable`. For group-by operations with `string` keys and `i32` values, the generated code declares:

```
// htable
int *t0_htable = malloc(...);
// Key-Value arrays
const char **t0_keys_str = malloc(...); // string keys pointer
int *t0_keys_len = malloc(...);         // string keys length
int32_t *t0_values = malloc(...);       // i32 values
int t0_key_count = 0;
```

This open-addressing hash table structure represents objects produced by group-by operations. Collision resolution uses linear probing. Each `htable` entry stores an index into the value arrays, where values are inserted sequentially in adjacent positions.

For nested group-by queries requiring hash tables as values, we represent nested hash tables as C structs while keeping top-level hash tables as local variable collections. These structs contain the same fields as top-level hash tables (htable, key-value arrays). The containing hash table stores pointers to heap-allocated structs, which are lazily allocated upon inserting new keys.

4.2 Fusing Temporary Variables

For Rhyme queries that contain stateful aggregations in path expressions, Rhyme's JS codegen will generate a fresh temporary for each aggregation. Consider the following example:

```
{ data.*.key: (sum data.*.value) / (count data.*.value) }
```

The query will have to maintain three temporaries: `t0` for the result object, `t1` and `t2` for the sum and count with respect to each unique `data.*.key`, resulting in the following generated code:

```
let t0, t1, t2; t0 ??= {}; t1 ??= {}; t2 ??= {}
for (let x in data) {
  t1[data[x].key] ??= 0; t2[data[x].key] ??= 0;
  t1[data[x].key] += data[x].value; t2[data[x].key] += 1
  t0[data[x].key] = t1[data[x].key] / t2[data[x].key] }
```

If we translate this directly into C, it requires three separate hash tables for the three temporaries and we will end up doing **eight** hash table lookups using the same key into the three tables. This leads to extremely inefficient C code especially on large datasets. We observe that this computation can be instead accomplished with only one hash table for the main result and separate value arrays for correlated aggregations in the path. In the C codegen, the two hash tables for `sum` and `count` will only emit their value arrays and share the same keys and `htable` with the root hash table:

```
// ... hash table fields for t0
double *t0_values = ...;
int32_t *t1_values = ...;    // Fused sum values
int32_t *t2_values = ...;  // Fused count values
```

In the actual loop body, all lookups are performed on a single hash table, and updates target different value arrays. However, excessive hash lookup/probing overhead still exists. Common subexpression elimination (CSE) is needed to remove redundant lookups because these hash tables are insertion-only and never delete entries, so lookup results can be safely reused. Performing CSE at this stage is difficult, though, because the final loop structure and control flow are not yet known. Two lookups might be merged incorrectly if their statements are later placed in different loops, violating C scoping rules.

As a result, we incorporate a lazy emission strategy for hash lookups. Hash-lookup CSE is delayed until loop scheduling, when program structure is fixed and we can safely determine which lookup results can be reused. In our running example, the generated code contains only one hash lookup, and all subsequent accesses reuse that result to index into the table, even though they originate from separate IR statements.

4.3 Optimizing CSV File Loading

Inefficient data movement would result in a significant performance bottleneck, given the gap between CPU processing speeds and slow

physical storage disks. For CSV files, the compiler leverages the provided schema to generate specialized parsing loops. To maximize parsing efficiency, we map columns directly into temporary local variables without materializing the parsed CSV file in memory, and the file is mapped into memory via `mmap`. This approach, inspired by the `StreamCSV` loading mode of Flare [21], generates the following loop for `* <- loadCSV(...)`:

```
size_t i = 0, n = fsize(fd);
for (int x = 0; i < n; x++) {    // Scan through one row
    while (file[i] != ',') i++;  // skip unused column
    int population = 0;
    while (file[i] != '\n') {    // get column population
        population = population * 10 + (file[i] - '0');
        i++; }
    ... /* loop body */ }
```

The parser maintains a byte-wise cursor through the file, identifying column boundaries via delimiters (commas and newlines for CSV, | for TBL format). Schema information guides column parsing: primitive types (decimals, integers) are parsed inline while advancing the cursor, whereas strings record only start and end indices. This strategy eliminates null-termination requirements and avoids string copies—all values remain in local variables. Column projection further optimizes parsing by extracting only referenced columns; unused columns are scanned and discarded.

4.4 Optimizing SQL-Style Queries

With the expressiveness of the Rhyme language, we can naturally write tabular queries that perform classic relational operations. However, translating such queries into C can still yield inefficient code compared with compiled database engines if done naively. For the SQL-style subset of Rhyme queries, we generate efficient C code matching the performance of state-of-the-art systems that compile SQL to native code, despite Rhyme's broader expressiveness.

We show a specialization that applies automatically to objects with compile-time-constant keys, which are lowered to columnar value arrays. The specialization triggers on the natural way to write a projection or grouped aggregation, and we examine its robustness to rephrasing below. Table 2 demonstrates common SQL-style queries in Rhyme using TPC-H [45] benchmark tables.

Projection and Group-By. Multi-column projection (Table 2 row 1) and group-by with multiple aggregations (row 4) both build per-record objects whose field names are compile-time constants. Rhyme's IR would encode these as nested group-bys with constant inner keys, which a naive translation would realize as nested hash tables. The backend recognizes this all-constant-keys pattern and instead expands the result into multiple parallel C arrays—a columnar layout in which record i is the tuple of elements at index i across the value arrays. This applies to both the array (projections) and hash table (group-bys) structures.

Hash Joins and Variants. Section 2 introduced joins via shared generator symbols, where the iteration ranges over keys present in both collections. That style fits hierarchical data keyed by the join column, but SQL-style joins typically combine tables on a *field* of the records rather than the outer key. For such joins, Rhyme expresses an explicit two-phase build/probe pattern: phase 1 builds a hash table on the join column via group-by over one table; phase 2 iterates the second table and probes. The region-nation join illustrates the base pattern:

Table 2: Common SQL queries expressed in Rhyme, different variations of join are explained in Section 4.4

Operation	Rhyme Query

Projection	`// Project the two columns`
	`[{ nkey: nation.*.n_nationkey,`
	`  name: nation.*.n_name }]`

Filtering	`// Get the record that has the name "ASIA"`
	`[region.*.r_name == "ASIA" &`
	`  { rkey: region.*.r_regionkey }]`

Aggregation	`// Count the number of line items` `count lineitem.*`

Group By	`// object notation & pipe, group part by type`
	`{`
	`  type: single part.*.p_type,`
	`  count: count part.*`
	`} | group part.*.p_type`

```
// Phase 1: build region.r_regionkey -> records
let regionMap = [{ name: region.*r.r_name }] |
                  group region.*r.r_regionkey
// Phase 2: probe with foreign key
[{ nation: nation.*n.n_name,
   region: regionMap.(nation.*n.n_regionkey).*r2.name }]
```

A failed lookup skips the inner loop, giving inner-join semantics. Two optimizations of this base pattern matter in practice.

Linked-bucket layout. The phase-1 hash table maps each key to multiple records. We represent this with a single flat value array plus auxiliary `head`/`prev` arrays forming per-key linked lists, ensuring sequential insertion even across keys and improving locality compared with per-key dynamic arrays. The `regionMap` produces:

```
int *tmp_head = malloc(/* hash size */);
int *tmp_prev = malloc(/* hash size x factor */);
const char **tmp_name = malloc(/* hash size x factor */);
```

Composite Group Keys. Multi-column `GROUP BY` would naively produce nested hash tables. We instead let the user write the group key as an array literal, which the backend lowers to a single hash table with parallel key arrays. When the user inserts into the hash table, a hash value will be calculated for the composite key. TPC-H Q1, which groups by two columns `l_returnflag` and `l_linestatus`, becomes:

```
{ /* ... aggregations ... */ } |
  group [lineitem.*.l_returnflag, lineitem.*.l_linestatus]
```

Robustness of the specialization. A natural concern with the automatic constant-key expansion is brittleness—that a small rewrite might fall outside the recognized form and silently lose performance. The expansion fires whenever an object's keys are compile-time constants, which is the idiomatic way to construct records, projections, and bundles of named aggregations; it is lost

only by constructing keys dynamically, the less natural phrasing. Within this regime the columnar layout makes structural choices free of cost. For instance, a group-by whose value is a single-field record `{ data.*A.k: { a: sum data.*A.v } }` stores the field `a` in the same value array that the bare form `{ data.*A.k: sum data.*A.v }` would use, rather than allocating a nested hash table; the wrapping object changes only how the result is read back (indexing `.a`), not how it is stored or computed. Likewise, a multi-valued grouping that collects an array of scalars and one that collects an array of constant-key objects share the same flat linked-bucket hash table, differing only in how many parallel value arrays are attached to each bucket.

5 EXPERIMENTS

In this section, we evaluate the performance of C code generated by Rhyme's C backend on benchmark queries covering different styles of workloads.

In the first experiment, we evaluate SQL-style queries over structured, tabular data using the standard TPC-H benchmark. We compare against PostgreSQL, CedarDB [15]—the commercial database system that builds on the Umbra [32] research database, carrying over its compilation-based execution stack—and Flare [21], which generates high-performance C code for Spark query plans.

Beyond SQL-style tabular workloads, we evaluate the C backend on JSONBench, where the input is newline-delimited JSON (NDJSON) containing semi-structured Bluesky events. We compare performance with PostgreSQL, DuckDB, and CedarDB.

Because JSONBench queries are limited in structure and do not fully cover hierarchical workloads, we add four additional JSON queries that perform nested group-bys over GitHub commit-history data from the Linux kernel repository. Finally, we present an ablation study that analyzes the impact of C-backend specializations and optimizations by disabling each in turn.

In general, our experiments show that the C code generation backend of Rhyme is able to accelerate Rhyme queries to the same performance level as the state-of-the-art compiled query engine that works with in-memory data on a single thread for both structured and semi-structured data.

Configurations. All queries are executed five times on a single core, and we report median runtime in milliseconds. We run each query once for warm-up. We disable parallelism in PostgreSQL, DuckDB, and CedarDB. We use the `preload` mode for both Flare and Rhyme to load the CSV file into memory first except for the additional nested queries.

Experimental Setup. All experiments are conducted on a single NUMA machine with 4 sockets, 24 Xeon(R) Platinum 8168 cores per socket, and 750GB RAM per socket. The operating system is Ubuntu 22.04.5 LTS. We use PostgreSQL 18.1 and CedarDB 2026-04-20. We use Node.js 20.19.5 for running the Rhyme compiler and GCC 11.4.0 for the compilation of the generated C code of both Rhyme and Flare with flag `-O3`. For JSONBench and the additional nested queries, we use DuckDB 1.4.4 and JQ 1.8.1.

5.1 TPC-H Runtime

In this experiment we compare the single-core performance of Rhyme's generated C code against PostgreSQL, CedarDB, and Flare on the TPC-H benchmark at scale factor 10. We run PostgreSQL without creating any auxiliary indexes. Flare executes the query plan generated by Spark. Because join order in Rhyme is written explicitly by the programmer rather than selected by a cost-based optimizer, we fix the Rhyme join orders by inspecting the plans that both Flare and CedarDB produce for all 22 queries. The two largely agree; where they differ, we write the Rhyme query to follow the faster plan. Q12, Q16, Q17, Q18, and Q19 adopt CedarDB's join order, and the rest follow the order shared by both systems. The absolute runtime for all queries is presented in Figure 2.

With matched plans, Rhyme tracks the C code generated by Flare closely. It is faster than Flare on 13 of the 22 queries and within 15% on another six; it is slower only on Q22 (2.1×), Q1 (1.5×), and marginally on Q12 (1.2×). These gains come largely from query-level optimizations—notably the reuse of shared subqueries—and, on Q17 and Q19, from the choice of plan. This demonstrates that Rhyme's more general and expressive query language does not compromise performance on structured tabular workloads.

Rhyme can additionally specialize a join on a unique key into a direct lookup—aggregating the build side with `single` yields a single-record-per-key table and elides the inner probe loop. This is a trade-off rather than an unconditional win: it suits chains of joins over unique keys, but when an early join is selective, flattening the nested loop into eager top-level lookups gives up short-circuiting, so each later probe runs for every row rather than only for rows that already matched.

Query-level optimization provides additional gains. Q15 (1.8×) illustrates this: the query computes the supplier with maximum total revenue from `lineitem` and joins back to `supplier` details. Flare's generated code scans `lineitem` twice: once to build the `l_suppkey` → `total_revenue` aggregation, then again to perform the join after computing the maximum. Rhyme constructs the aggregate once and reuses it for both operations, halving the `lineitem` passes. This optimization is a direct consequence of Rhyme's query language allowing natural expression of shared subqueries, whereas Spark's logical planner does not identify this opportunity.

Q17 and Q19 show the value of choosing the plan explicitly. Under Flare's plan, both materialize the 60M-row `lineitem` table into multi-valued linked-bucket hash tables keyed by the non-unique `l_partkey`, and chasing those bucket chains dominates runtime—a regression in an earlier version that followed Flare's plan. CedarDB's plan instead filters the smaller side first and probes: for Q17 the generated code builds a set from the filtered `part` table and probes `lineitem`, reducing two multi-valued bucket tables to one, and for Q19 it removes the linked-bucket tables entirely. Following these plans, Rhyme is 5.3× (Q17) and 3.5× (Q19) faster than Flare's plan-bound code.

A property of the hash tables also works against Rhyme. Flare's join hash table is merely a join index—build rows are appended to a per-position chain and keys are checked at probe time, so distinct keys colliding to a position share a chain and are never resolved into separate slots. Rhyme's hash table is the materialized JSON object, so each distinct key needs its own slot; it uses open-addressing with linear probing, paying collision resolution on every insert and update—necessary for object semantics but costlier than Flare's append-and-chain on high-collision, non-unique groupings.

TPC-H SF10	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10	Q11	Q12	Q13	Q14	Q15	Q16	Q17	Q18	Q19	Q20	Q21	Q22
PostgreSQL	50635	5791	29481	20123	24027	6191	15066	9429	50736	17216	1980	12788	19913	7591	15878	6451	56485	96035	9733	24949	35142	14494
CedarDB	3195	101	612	447	682	82	493	462	2572	1100	114	312	5316	262	262	435	206	1637	162	255	1449	345
Flare	376	241	1026	1027	1144	226	1297	1627	4358	2181	128	1013	2882	225	384	948	3440	1544	1090	1676	3123	338
Rhyme	554	162	561	1056	608	201	775	519	1598	1387	35	1174	2751	220	208	675	655	1178	310	1627	3538	723

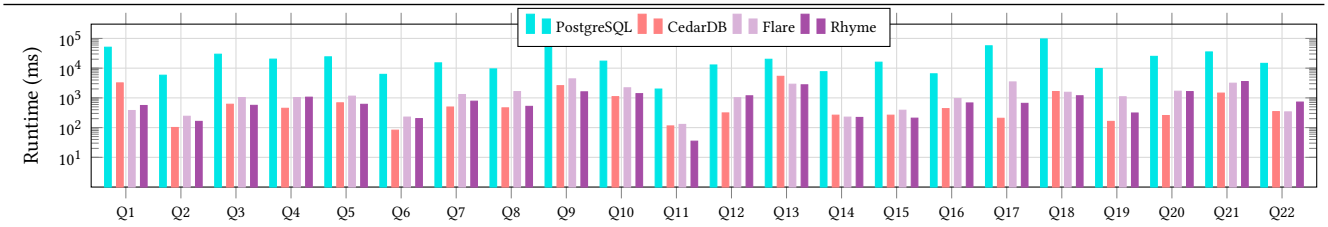


Figure 2: Single-thread absolute runtime in milliseconds (ms) of TPC-H queries with SF10 for PostgreSQL, CedarDB, generated C code of Flare and generated C code of Rhyme. Data loading time is excluded.

Compared with CedarDB—a compiling DBMS with a cost-based optimizer—the results are mixed. CedarDB is faster on Q20 (6.4×), Q12 (3.8×), and Q17 (3.2×), and on Q4, Q6, Q21, and Q22 by 2–2.5×. Rhyme is faster on scan-heavy or simple-group queries: Q1 (5.8×), Q11 (3.3×), Q13 (1.9×), and Q9 (1.6×), and is competitive on the rest. We adopted CedarDB's join order for several queries, but its generated machine code and internal data structures are not available to us, so we describe what Rhyme's own generated code does rather than attributing the gap to CedarDB's execution.

On the queries where CedarDB leads, Rhyme's generated code materializes `lineitem` or `partsupp` into multi-valued linked bucket hash tables over a non-unique key, even under CedarDB's join order: Q20 groups `lineitem` by `l_partkey` and `l_suppkey`, and Q17 retains one bucket table on `l_partkey`. Building these buckets and chasing their chains over tens of millions of rows is the dominant cost in Rhyme's code. Rhyme's wins lie in the opposite regime: Q1 is a single `lineitem` scan with a composite-key flat-array hash table inlined into the loop, and Q11 is a compact multi-way join with small hash tables—queries with little planning headroom, where statically specialized code runs close to the metal.

In summary, Rhyme's generated C code matches or exceeds Flare across most of TPC-H and is competitive with CedarDB, despite expressing a far more general query language. We also observe subtle runtime differences even when join orders are identical: Rhyme may emit loops in a different order, which can be less cache-friendly and therefore affect performance. Against a compiled engine, this is a workload-dependent trade-off rather than a blanket speedup.

5.2 JSONBench Runtime

We evaluate Rhyme on JSONBench, a benchmark for querying semi-structured JSON data from Bluesky social network events containing 10 million records. We compare the query runtime with PostgreSQL, DuckDB, and CedarDB. For PostgreSQL and CedarDB, data is pre-loaded into a table with a single JSONB column; for DuckDB, we use a JSON column. We measure query execution time only, excluding table scan and disk I/O overhead. As shown in Figure 3, Rhyme outperforms PostgreSQL, DuckDB, and CedarDB across all five queries.

JSONBench	Q1	Q2	Q3	Q4	Q5
PostgreSQL	3150	187876	12562	16583	16983
DuckDB	4700	10870	8580	1700	2230
CedarDB	4435	8320	5883	4841	4856
Rhyme	1355	4497	2343	1626	1717

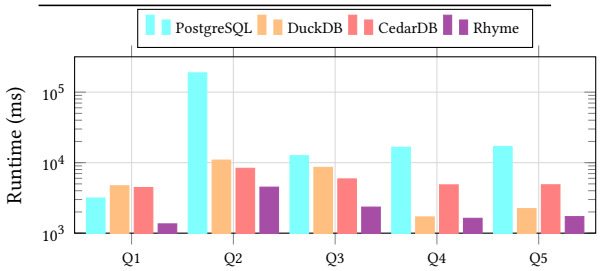


Figure 3: Single-thread absolute runtime in milliseconds (ms) of JSONBench queries with 10 million rows of data for PostgreSQL, DuckDB, CedarDB and generated C code of Rhyme. Data loading time is excluded.

The JSONBench queries perform single-pass aggregations with filtering and grouping on nested JSON fields, including basic aggregations (Q1), `COUNT DISTINCT` with grouping (Q2), timestamp extraction and temporal grouping (Q3), and per-user min/max aggregates (Q4–Q5). Against PostgreSQL, Rhyme achieves speedups ranging from 2.3× (Q1) to 42× (Q2), with typical speedups of 5×–10× on the remaining queries. Against DuckDB, speedups range from 1.3× (Q5) to 3.7× (Q3). Against CedarDB, speedups range from 1.9× (Q2) to 3.3× (Q1).

DuckDB uses the same YYJSON [8] library as Rhyme to parse and navigate JSON, so that cost is shared—Rhyme specializes neither the parsing nor the object/array access. The advantage is in query execution: DuckDB runs a (vectorized) interpreter, whereas Rhyme emits code specific to the query, specializing the aggregation hash tables to the statically known key and value types, so they are tagless and concretely typed (Section 4). Avoiding the interpretation and dynamic-typing overhead this way is the primary source of Rhyme's 1.3–3.7× advantage over DuckDB. PostgreSQL, likewise interpreted, serves only as a baseline.

Nested Queries	Q1	Q2	Q3	Q4
JQ	71315	71907	65650	65486
Rhyme (JS)	49220	45291	45573	42612
Rhyme (C)	4085	4392	3934	3837

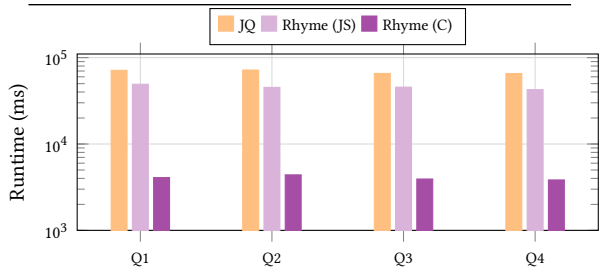


Figure 4: Single-thread absolute runtime in milliseconds (ms) of additional nested queries with 1 million GitHub commit objects for JQ, Rhyme with the JS backend and Rhyme with the C backend. Data loading included.

Disabled optimization	Query	Full	Ablated	Slowdown
Const key columnar → nested map	TPC-H Q1*	550	33898	61.6×
Hash-lookup CSE	TPC-H Q1	554	1017	1.8×
Hash-lookup CSE	JSONBench Q2	4497	6316	1.4×
Bucket layout → nested arrays	TPC-H Q8	519	1032	2.0×

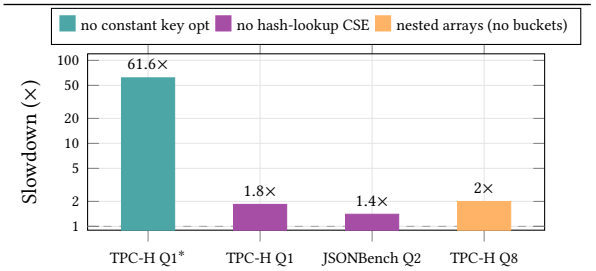


Figure 5: Ablation: slowdown (log scale) when each optimization is disabled, normalized to the full system (dashed line at 1×). Color indicates the disabled optimization. *Q1 variant that omits the average aggregates and carries additional constant-key fields, to expose the cost difference.

CedarDB is the strongest baseline here—faster than PostgreSQL on every query and DuckDB on most. It represents JSON using JSON Tiles [20], which detects frequently-occurring keys across the document collection and materializes them into a columnar layout, with a parsed fallback for irregular fields, so that scans over the hot keys approach relational performance. Even so, Rhyme is 1.9–3.3× faster. We lack CedarDB's execution breakdown and leave a precise attribution to future work; on Rhyme's side the whole query is a single fused scan specialized to that query—YYJSON `READ_INSITU` parsing with no string copies, and aggregation hash tables inlined into the loop.

The results show that Rhyme excels in querying large nested JSON datasets and is able to outperform both traditional relational systems with JSON support and modern compiled engines that target heterogeneous data.

5.3 Additional Nested Queries Runtime

While JSONBench queries produce flat tabular results, we designed four additional queries to evaluate Rhyme's performance on hierarchical nested aggregations. These queries operate on one million GitHub commit objects from the Linux kernel repository [4], organized as an NDJSON file of 10,000 lines with 100 objects per line. The queries perform up to three levels of nested group-by operations with aggregations at each level (e.g., commits per author, per year, and per month), producing deeply nested JSON output. We compare Rhyme with JQ and include end-to-end timing with data loading, since JQ does not isolate loading and parsing time.

As shown in Figure 4, Rhyme's C backend achieves 16×–17× speedup over JQ and 10×–12× over Rhyme's JS backend across all four queries. The performance gap is attributable to fundamental execution-model differences. Rhyme's generated C code performs a single scan using nested hash tables for aggregations with in-place updates. String pointers reference the original input buffer via YYJSON's `READ_INSITU` mode, eliminating string copies. By contrast, JQ materializes all JSON objects into its internal format, generating substantial allocation pressure.

In addition, JQ's `group_by` construct materializes intermediate arrays at each nesting level: the input is first grouped by author, then each author's commits are re-grouped by year, and finally each year's commits are re-grouped by month. This multi-pass approach requires allocating and traversing intermediate data structures, compounded by JQ's interpreter overhead and full materialization of JSON objects into its internal representation.

The JS backend outperforms JQ in all queries. When we look at the time more closely we see that the majority of the execution time is spent in data loading, with actual query execution under 10 seconds. This overhead arises from `JSON.parse` constructing full V8 object trees for each NDJSON line, generating heap allocations for every field and creating sustained garbage collection pressure.

In summary, both backends of Rhyme outperform JQ on queries that produce nested results, showing the benefit of specialized C code generated for queries.

5.4 Ablation Study

The performance stems not only from native code generation but from a handful of specializations (described in Section 4). To attribute the gains to specific mechanisms, we ablate three of these optimizations: the lazy hash-lookup CSE, the flat group-by linked bucket layout, and the constant-key columnar layout by disabling each independently, with all else held fixed, and measuring the slowdown on representative queries (Figure 5).

We first disable the hash-lookup CSE, which reuses a single computed slot index across all accesses to the same key within a loop body. We test it on two queries that probe the same hash entry many times per record: TPC-H Q1, with several per-group aggregations, and JSONBench Q2, a per-group `COUNT DISTINCT`. With the CSE, Q1's loop body contains a *single* hash-table probe whose slot is reused for all of the group's reads and updates; disabling it expands Q1 to 21 separate probes and Q2 to 14, as each access recomputes the hash and re-runs the probe loop, slowing Q1 by 1.8× (554 → 1017 ms) and Q2 by 1.4× (4497 → 6316 ms).

We next ablate the storage for the group-by-array pattern (a multi-valued grouping over a non-unique key, Section 4). The default flat layout, a single shared value array with per-key linked buckets, is replaced by heap-allocated nested arrays. On TPC-H Q8, which builds three such groupings (on `l_orderkey`, `o_custkey`, and `l_suppkey`), the extra per-key allocation and the pointer dereference on every access slow the query from 519 to 1032 ms (2.0×), confirming the value of the contiguous bucket layout.

Finally, we disable the constant-key columnar optimization (Section 4). A record whose keys are all compile-time constants—e.g. Q1's per-group `{sum_qty, sum_base_price, ...}`—is normally split into one value array *per key*; because the constant keys are distinct, each array holds a single, uniform type and the record stays flat. Disabling it degrades the record on two fronts: it becomes a nested hash table keyed by the field names and its single value array must now hold fields of differing types, which forces a value array of *dynamic tagged* values, adding an overhead due to runtime type dispatch on every read. To expose the gap, we use a Q1 variant (Q1*) that drops the average aggregates and carries additional constant-key fields, so more values flow through the record each row. The combined effect is by far the largest we measure: 550 → 33898 ms (61.6×), showing that the columnar split is what keeps per-group state both flat and tag-less.

Summary. Our evaluation demonstrates that Rhyme's C code generation backend achieves state-of-the-art performance across structured, semi-structured, and hierarchical workloads through multiple benchmarks—showing that its expressive query model supporting nested data does not compromise efficiency on traditional SQL workloads. An ablation study further attributes this performance to the backend's key specializations, each of which yields a measurable slowdown when disabled.

6 RELATED WORK

Compiled Query Engines. The data-centric compilation model of HyPer [25, 31] established that compiling query plans to machine code via LLVM [29] can dramatically outperform interpreted execution, with later variants using template-based codegen [28] and compilation integrated into an OLTP engine [19]; MonetDB/X100 [14] pursues vectorized execution as an orthogonal alternative. Umbra [32] advances HyPer with a custom IR and a fast compilation backend [26]. A framework-based line derives compilers from high-level interpreters via LMS [39]—LegoBase [27], LB2 [44], the SQL-to-C pearl [38], and DBLAB [42]; Flare [21], a Spark accelerator on LMS that emits native C, directly inspired Rhyme's C backend.

Trends in SQL and Query Language Design. Despite SQL's status as the de-facto standard for relational data, it has faced sustained criticism for readability and composability [33]. Recent efforts focus on extending rather than replacing it: Google's pipe syntax [43] adds left-to-right data flow to SQL and has been adopted in GoogleSQL [13]. PRQL [6] offers an alternative pipelined syntax that compiles to SQL. SQL++ [34] extends SQL with richer support for nested data types. AsterixDB [11], designed for semi-structured data, employs AQL and SQL++ as its query languages.

Rhyme takes a different angle: rather than patching SQL, it provides a fresh declarative model where query structure mirrors result structure, with native support for nested data and a pipe operator for sequential transformations, while delivering SQL-level performance on relational workloads.

Query Languages and Engines for Semi-Structured Data. A rich ecosystem of languages and systems addresses querying beyond flat tables. XQuery [7] and JSONiq [3, 37] are standardized languages for XML and JSON; Rumble [30] scales JSONiq to large datasets via Spark. JQ [2] is a widely-used stream-oriented JSON processor, and JSONPath [22] provides path-based selection in JSON documents, which directly inspired Rhyme's path expression syntax. GraphQL [1] targets hierarchical API responses. On the analytics performance side, JSON Tiles [20] automatically detects frequently-accessed keys in schemaless JSON and extracts them into columnar storage, achieving near-relational scan performance while preserving schema flexibility. Closest to our performance goals, Proteus [24] JIT-compiles queries over heterogeneous relational and nested data via LLVM, but exposes a relational operator interface rather than a structure-mirroring one. SDQL [41] is a functional collection language built on semi-ring dictionaries that likewise compiles queries over flat and nested data.

Rhyme's declarative group-by model is designed to naturally express the hierarchical aggregations these languages support, while the C backend generates code competitive with native query engines on both flat and nested workloads.

7 CONCLUSION

We have presented a native C code generation backend for Rhyme that demonstrates a key result: a declarative query language expressive enough to naturally handle nested, hierarchical data does not need to sacrifice performance on traditional relational workloads. We establish Rhyme as a single language that expresses both relational and hierarchical queries naturally: refined generator-binding and missing-value semantics that capture co-iteration, nested loops, and inner/outer joins; and a dependency-driven scheduler based on fine-grained per-statement constraints that correctly synthesizes loop structure from a loop-free, branch-free IR. Two further contributions make this expressiveness efficient: a gradual type system, expressed in the same object notation as queries, that eliminates tagged-union overhead and drives the selection of specialized internal data structures; and a suite of backend specializations including a columnar layout for constant-key objects, fused temporaries with lazy hash-lookup CSE, linked-bucket hash tables for multi-valued groupings, composite group-by keys, and zero-copy CSV parsing.

The evaluation shows this across structured, semi-structured, and hierarchical workloads: Rhyme matches or outperforms state-of-the-art compiled SQL engines on TPC-H and outpaces JSON-capable databases and tools on nested queries. It fills a gap left by existing systems—relational engines lack natural support for hierarchical queries, while JSON-oriented tools lack the performance needed for SQL-style analytics—handling both in a single declarative language at no measured performance cost on either.

ACKNOWLEDGMENTS

We thank Ruiqi Gao, Isaac Fleetwood, and Supun Abeysinghe for their contributions to critical components including the type system and loop scheduling of the Rhyme project during their time at Purdue University.

REFERENCES

[1] [n.d.]. *GraphQL: A query language for APIs.* Retrieved 2026-02-27 from https://graphql.org/

[2] [n.d.]. *jq manual.* Retrieved 2026-02-27 from https://jqlang.github.io/jq/manual/

[3] [n.d.]. *JSONiq.* Retrieved 2026-02-27 from https://www.jsoniq.org/

[4] [n.d.]. *Linux Kernel.* Retrieved 2026-02-27 from https://github.com/torvalds/linux/

[5] [n.d.]. *PostgreSQL: The World's Most Advanced Open Source Relational Database.* Retrieved 2026-02-27 from https://www.postgresql.org/

[6] [n.d.]. *PRQL: Pipelined Relational Query Language.* Retrieved 2026-02-27 from https://prql-lang.org/

[7] [n.d.]. *XQuery 3.1: An XML Query Language.* Retrieved 2026-02-27 from https://www.w3.org/TR/xquery-31/

[8] [n.d.]. *yyjson doc.* Retrieved 2026-02-27 from https://ibireme.github.io/yyjson/doc/doxygen/html/api.html

[9] Supun Abeysinghe and Tiark Rompf. 2024. Rhyme: A Data-Centric Expressive Query Language for Nested Data Structures. In *Practical Aspects of Declarative Languages: 26th International Symposium, PADL 2024, London, UK, January 15–16, 2024, Proceedings* (London, United Kingdom). Springer-Verlag, Berlin, Heidelberg, 64–81. https://doi.org/10.1007/978-3-031-52038-9_5

[10] Supun Abeysinghe and Tiark Rompf. 2024. Rhyme: A Data-Centric Multi-paradigm Query Language Based on Functional Logic Metaprogramming: System Description. In *Functional and Logic Programming: 17th International Symposium, FLOPS 2024, Kumamoto, Japan, May 15–17, 2024, Proceedings* (Kumamoto, Japan). Springer-Verlag, Berlin, Heidelberg, 273–288. https://doi.org/10.1007/978-981-97-2300-3_14

[11] Sattam Alsubaiee, Yasser Altowim, Hotham Altwaijry, Alexander Behm, Vinayak Borkar, Yingyi Bu, Michael Carey, Inci Cetindil, Madhusudan Cheelangi, Khurram Faraaz, Eugenia Gabrielova, Raman Grover, Zachary Heilbron, Young-Seok Kim, Chen Li, Guangqiang Li, Ji Mahn Ok, Nicola Onose, Pouria Pirzadeh, Vassilis Tsotras, Rares Vernica, Jian Wen, and Till Westmann. 2014. AsterixDB: a scalable, open source BDMS. *Proc. VLDB Endow.* 7, 14 (Oct. 2014), 1905–1916. https://doi.org/10.14778/2733085.2733096

[12] 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. https://doi.org/10.1145/2723372.2742797

[13] David F. Bacon, Nathan Bales, Nico Bruno, Brian F. Cooper, Adam Dickinson, Andrew Fikes, Campbell Fraser, Andrey Gubarev, Milind Joshi, Eugene Kogan, Alexander Lloyd, Sergey Melnik, Rajesh Rao, David Shue, Christopher Taylor, Marcel van der Holst, and Dale Woodford. 2017. Spanner: Becoming a SQL System. In *Proceedings of the 2017 ACM International Conference on Management of Data* (Chicago, Illinois, USA) *(SIGMOD '17)*. Association for Computing Machinery, New York, NY, USA, 331–343. https://doi.org/10.1145/3035918.3056103

[14] Peter A. Boncz, Marcin Zukowski, and Niels Nes. 2005. MonetDB/X100: Hyper-Pipelining Query Execution. In *CIDR*.

[15] CedarDB. 2025. *CedarDB.* Retrieved 2026-02-27 from https://cedardb.com Commercial database system based on the Umbra research database.

[16] S. Ceri, G. Gottlob, and L. Tanca. 1989. What You Always Wanted to Know About Datalog (And Never Dared to Ask). *IEEE Trans. on Knowl. and Data Eng.* 1, 1 (March 1989), 146–166. https://doi.org/10.1109/69.43410

[17] E. F. Codd. 1979. Extending the database relational model to capture more meaning. *ACM Trans. Database Syst.* 4, 4 (Dec. 1979), 397–434. https://doi.org/10.1145/320107.320109

[18] Alain Colmerauer and Philippe Roussel. 1996. The birth of Prolog. *History of programming languages—II* (1996), 331–367.

[19] Cristian Diaconu, Craig Freedman, Erik Ismert, Per-Ake Larson, Pravin Mittal, Ryan Stonecipher, Nitin Verma, and Mike Zwilling. 2013. Hekaton: SQL server's memory-optimized OLTP engine. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data* (New York, New York, USA) *(SIGMOD '13)*. Association for Computing Machinery, New York, NY, USA, 1243–1254. https://doi.org/10.1145/2463676.2463710

[20] Dominik Durner, Viktor Leis, and Thomas Neumann. 2021. JSON Tiles: Fast Analytics on Semi-Structured Data. In *SIGMOD*. 445–458.

[21] Grégory M. Essertel, Ruby Y. Tahboub, James M. Decker, Kevin J. Brown, Kunle Olukotun, and Tiark Rompf. 2018. Flare: optimizing apache spark with native compilation for scale-up architectures and medium-size data. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation* (Carlsbad, CA, USA) *(OSDI'18)*. USENIX Association, USA, 799–815.

[22] Stefan Goessner. [n.d.]. *JSONPath - XPath for JSON.* Retrieved 2026-02-27 from https://goessner.net/articles/JsonPath/

[23] John Grant. 2008. Null values in SQL. *SIGMOD Rec.* 37, 3 (Sept. 2008), 23–25. https://doi.org/10.1145/1462571.1462575

[24] Manos Karpathiotakis, Ioannis Alagiannis, and Anastasia Ailamaki. 2016. Fast queries over heterogeneous data through engine customization. *Proc. VLDB Endow.* 9, 12 (Aug. 2016), 972–983. https://doi.org/10.14778/2994509.2994516

[25] Alfons Kemper and Thomas Neumann. 2011. HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots. In *2011 IEEE 27th International Conference on Data Engineering*. 195–206. https://doi.org/10.1109/ICDE.2011.5767867

[26] Timo Kersten, Viktor Leis, and Thomas Neumann. 2021. Tidy Tuples and Flying Start: Fast Compilation and Fast Execution of Relational Queries in Umbra. *VLDB J.* 30, 5 (2021), 883–905.

[27] Yannis Klonatos, Christoph Koch, Tiark Rompf, and Hassan Chafi. 2014. Building Efficient Query Engines in a High-Level Language. In *VLDB*, Vol. 7. 853–864.

[28] Konstantinos Krikellas, Stratis D. Viglas, and Marcelo H. Cintra. 2010. Generating Code for Holistic Query Evaluation. In *ICDE*. IEEE, 613–624.

[29] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization* (Palo Alto, California) *(CGO '04)*. IEEE Computer Society, USA, 75.

[30] Ingo Müller, Ghislain Fourny, Stefan Irimescu, Can Berker Cikis, and Gustavo Alonso. 2020. Rumble: Data Independence for Large Messy Data Sets. *Proc. VLDB Endow.* 14, 4 (2020), 498–506.

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

[32] Thomas Neumann and Michael Freitag. 2020. Umbra: A Disk-Based System with In-Memory Performance. *CIDR* (2020).

[33] Thomas Neumann and Viktor Leis. 2024. A Critique of Modern SQL And A Proposal Towards A Simple and Expressive Query Language. *CIDR* (2024).

[34] Kian Win Ong, Yannis Papakonstantinou, and Romain Vernoux. 2015. The SQL++ Query Language: Configurable, Unifying and Semi-structured. arXiv:1405.3631 [cs.DB] https://arxiv.org/abs/1405.3631

[35] 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. https://doi.org/10.1145/3299869.3320212

[36] Wilmer Ricciotti and James Cheney. 2022. A Formalization of SQL with Nulls. *J. Autom. Reason.* 66, 4 (Nov. 2022), 989–1030. https://doi.org/10.1007/s10817-022-09632-4

[37] Jonathan Robie, Daniela Florescu, Ghislain Fourny, Matthias Brantner, and Donald Kossmann. 2012. The JSONiq Language. In *17th International Conference on Database Systems for Advanced Applications (DASFAA)*.

[38] Tiark Rompf and Nada Amin. 2015. Functional pearl: a SQL to C compiler in 500 lines of code. *SIGPLAN Not.* 50, 9 (Aug. 2015), 2–9. https://doi.org/10.1145/2858949.2784760

[39] Tiark Rompf and Martin Odersky. 2010. Lightweight modular staging: a pragmatic approach to runtime code generation and compiled DSLs. In *Proceedings of the Ninth International Conference on Generative Programming and Component Engineering* (Eindhoven, The Netherlands) *(GPCE '10)*. Association for Computing Machinery, New York, NY, USA, 127–136. https://doi.org/10.1145/1868294.1868314

[40] Claude Rubinson. 2007. Nulls, three-valued logic, and ambiguity in SQL: critiquing date's critique. *SIGMOD Rec.* 36, 4 (Dec. 2007), 13–17. https://doi.org/10.1145/1361348.1361350

[41] Amir Shaikhha, Mathieu Huot, Jaclyn Smith, and Dan Olteanu. 2022. Functional collection programming with semi-ring dictionaries. *Proc. ACM Program. Lang.* 6, OOPSLA1, Article 89 (April 2022), 33 pages. https://doi.org/10.1145/3527333

[42] Amir Shaikhha, Yannis Klonatos, Lionel Parreaux, Lewis Brown, Mohammad Dashti, and Christoph Koch. 2016. How to Architect a Query Compiler. In *Proceedings of the 2016 International Conference on Management of Data* (San Francisco, California, USA) *(SIGMOD '16)*. Association for Computing Machinery, New York, NY, USA, 1907–1922. https://doi.org/10.1145/2882903.2915244

[43] Jeff Shute, Shannon Bales, Matthew Brown, Jean-Daniel Browne, Brandon Dolphin, Romit Kudtarkar, Andrey Litvinov, Jingchi Ma, John Morcos, Michael Shen, David Wilhite, Xi Wu, and Lulan Yu. 2024. SQL Has Problems. We Can Fix Them: Pipe Syntax In SQL. *Proc. VLDB Endow.* 17, 12 (Aug. 2024), 4051–4063. https://doi.org/10.14778/3685800.3685826

[44] 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. https://doi.org/10.1145/3183713.3196893

[45] Transaction Processing Performance Council. 2008. TPC-H Benchmark. *Published at http://www.tcp.org/hspec. html* (2008).

[46] Ron van der Meyden. 1998. *Logical approaches to incomplete information: a survey.* Kluwer Academic Publishers, USA, 307–356.