DOI: 10.15514/ISPRAS-2025-37(6)-21

Beyond LLVM: Evaluating Fast Code Generation Alternatives for Query Compilation in PostgreSQL

M.V. Pantilimonov, ORCID: 0000-0003-2277-7155 <pantlimon@ispras.ru>
R.A. Buchatskiy, ORCID: 0000-0001-8522-1811 <ruben@ispras.ru>
D.V. Zavedeev, ORCID: 0009-0009-1477-5249 <zdenis@ispras.ru>

Ivannikov Institute for System Programming of the Russian Academy of Sciences,
25, Alexander Solzhenitsyn st., Moscow, 109004, Russia.

Abstract. The evolution of query compilation in database management systems traces back to System R, which pioneered a code generation scheme where small machine code fragments were stitched together to form a specialized routine to process a given SQL statement. Subsequent approaches shifted to generating C code, compiling it with system compilers like GCC into dynamic libraries, and loading them at runtime. The current state-of-the-art standard for dynamic query compilation is the LLVM framework, which bypasses frontend compiler overhead by directly generating intermediate representation, enabling machine-independent optimizations and efficient machine code generation. LLVM's resource-intensive nature, primarily designed as an optimizing compiler, however, can lead to compilation times that are orders of magnitude longer than query execution times, particularly problematic for queries with millisecond-level interpretation costs. This paper evaluates two lightweight code generation frameworks for x86-64 architecture as alternatives to LLVM in PostgreSQL, assessing their code generation speed and the quality of emitted machine code. We present a qualitative comparison with LLVM, analyzing trade-offs between compilation latency and runtime performance across databases of varying sizes. Experimental results demonstrate that lightweight code generation can not only outperform LLVM on small-scale datasets but also maintain competitive performance on larger ones.

Keywords: dynamic compilation; JIT-compilation; query execution; DBMS; PostgreSQL; LLVM; DynASM; AsmJIT.

For citation: Pantilimonov M.V., Buchatskiy R.A., Zavedeev D.V. Beyond LLVM: Evaluating Fast Code Generation Alternatives for Query Compilation in PostgreSQL. Trudy ISP RAN/Proc. ISP RAS, vol. 37, issue 6, part 2, 2025, pp. 77-92. DOI: 10.15514/ISPRAS-2025-37(6)-21.

Не LLVM единым: Исследование альтернативных методов быстрой генерации кода для компиляции запросов в PostgreSQL

М.В. Пантилимонов, ORCID: 0000-0003-2277-7155 <pantlimon@*ispras.ru*>
Р.А. Бучацкий, ORCID: 0000-0001-8522-1811 <ruben@ispras.ru>
Д.В. Заведеев, ORCID: 0009-0009-1477-5249 <zdenis@ispras.ru>

Институт системного программирования им. В.П. Иванникова РАН,
Россия, 109004, г. Москва, ул. А. Солженицына, д. 25.

Аннотация. Идея компиляции запросов в системах управления базами данных берёт своё начало в System R, где впервые была реализована схема генерации кода, при которой небольшие фрагменты машинного кода объединялись вместе для создания специализированной подпрограммы, обрабатывающей конкретный SQL запрос. В дальнейшем подходы изменились: вместо машинного кода начали генерировать код на языке C, который затем компилировался с помощью системных компиляторов, таких как GCC, в динамические библиотеки и подгружался в процессе выполнения. Сегодня стандартом де-факто в области динамической компиляции запросов стал фреймворк LLVM. Благодаря своей модульной архитектуре он позволяет избежать дорогостоящего этапа трансляции с языка высоко уровня в промежуточное представление, обеспечивая его прямую генерацию с последующим применением машинно-независимых оптимизаций и генерации эффективного машинного кода. Однако LLVM изначально разрабатывался как оптимизирующий компилятор, и его использование может приводить к значительным накладным расходам на компиляцию – в отдельных случаях они превышают время выполнения запроса в десятки раз. Это особенно проблематично для коротких запросов с миллисекундным временем исполнения. В данной работе рассматриваются два легковесных генератора кода для архитектуры x86-64 в качестве альтернативы LLVM в СУБД PostgreSQL. Оцениваются как скорость генерации кода с использованием этих фреймворков, так и качество получаемого исполняемого кода. Приведено качественное сравнение с LLVM, анализируются компромиссы между скоростью компиляции и производительностью выполнения запросов на базах данных различного размера. Результаты экспериментов показывают, что легковесные решения не только превосходят LLVM по производительности на небольших наборах данных, но и сохраняют её конкурентноспособной на больших объёмах информации.

Ключевые слова: динамическая компиляция; JIT-компиляция; выполнение запросов; СУБД; база данных PostgreSQL; платформа LLVM; компилятор DynASM; компилятор AsmJIT.

Для цитирования: Пантилимонов М.В., Бучацкий Р.А., Заведеев Д.В. Не LLVM единым: Исследование альтернативных методов быстрой генерации кода для компиляции запросов в PostgreSQL. Труды ИСП РАН, том 37, вып. 6, часть 2, 2025 г., стр. 77–92 (на английском языке). DOI: 10.15514/ISPRAS–2025–37(6)–21.

1. Introduction

Modern database management systems (DBMS) process queries through a multi-stage pipeline, beginning with the translation of a declarative query into a logical query plan represented as a tree of relational algebra operators. This logical plan is then converted into physical execution plan, which specifies data access methods and join algorithms. PostgreSQL DBMS [1] is a textbook example of this architecture, using the well-known iterator model, a.k.a. Volcano model [2], where query execution follows a pull-based approach – data flows from leaf nodes upward to the root, with each operator recursively requesting tuples from its children. While newer push-based execution model, popularized by Hyper [3], has demonstrated better performance – even when retrofitted onto Volcano-based execution engine not originally designed for it [4] – adopting such an approach would require a fundamental redesign of both the execution engine and the query planner. This transition would also introduce particular complexities in handling of certain operations like limit and merge joins [5]. Consequently, PostgreSQL continues to rely on iterator-based model, though substantial optimizations in expression evaluation have been implemented over the years. Version 10 introduced a bytecode-based interpreter [6-8] with direct threading technique that minimizes

dispatch overhead while improving branch prediction and cache locality. Building upon this foundation, PostgreSQL 11 added LLVM-based JIT [9] compilation for expressions [10], further accelerating analytical workloads. However, JIT compilation does not come for free – while it improves execution speed, the compilation overhead might outweigh these performance gains. This is particularly problematic with LLVM, which is fundamentally an optimizing compiler rather than a lightweight JIT engine. Its optimization pipeline and machine code emission are inherently expensive [11], making it almost certain to degrade performance when processing small datasets – cases where interpretation would be faster.

The most straightforward approach to decide whether to apply JIT compilation is to rely on the DBMS planner's cost estimation before query execution. However, even with perfect statistics and the most accurate cost models, these estimations will inevitably be wrong for some queries. Every such misprediction forces the system to pay full compilation costs for queries that would have executed faster via interpretation. To address this, multi-tier JIT compilation offers a promising direction, adopted in browser engines and virtual machines to balance startup latency and peak performance. Prior research [12] has explored system design with multi-tier compilation in mind, starting with interpretation and later switch to optimized LLVM-generated code after reaching a certain execution threshold. Alternatively, interpretation can be bypassed entirely, generating unoptimized machine code first [13] and then transitioning to optimized LLVM code compiled in parallel.

While such design performs well on server-grade hardware with abundant CPU cores, it becomes less optimal on commodity hardware. In more modest systems, dedicating resources to background compilation may actually reduce overall throughput. These computational resources could instead be used to serve additional concurrent queries, prioritizing system-wide throughput over single-query latency. The issue with (multi-tier) JIT in database systems lies in the low reusability of compiled query code, in contrast to browser engines and VMs. While one could implement a compiled query cache, the bookkeeping overhead is far from negligible and introduces significant challenges:

- Cache management: deciding which queries to cache, when to evict them and how to efficiently retrieve them. Cache invalidation may occur due to statistics updates or schema changes.
- Key selection: determining the cache key – should it be based on the query text, AST, execution plan, IR, etc. The key generation process is computationally expensive regardless of the chosen approach.
- Handling constants: optimize code with literals i.e. "colA > 5" or replace literal with variable i.e. "colA > :x" to reuse compiled queries for different values. This is a trade-off between the number of compiled queries and plan quality due to varying value distributions.
- Handling memory addresses: absolute addresses used in compiled code must be replaced with relative addresses or patched before execution. This introduces either extra metadata to track address locations needing patching along with a patching step or requires careful memory management and relative address calculations.

In contrast, JIT-compiled code in VMs or browser engines executes naturally when control flow reaches it, requiring minimal bookkeeping. Depending on compilation latency, it may be simpler and more efficient to compile queries on-demand each time rather than managing a query compilation cache.

We believe that the optimal solution lies in a JIT compiler with extremely low compilation latency, ensuring it almost never increases overall query execution time. For the simplest queries, however, interpretation should still be preferred, as it remains faster than even the most minimal compilation pipeline.

In our research, the overall goal is to assess whether lightweight JIT compilers can achieve performance comparable to LLVM-based JIT compilers in DBMS environments, without

introducing excessive compilation latency due to inaccurate query cost estimation. PostgreSQL serves as an ideal candidate for this experiment, as it already has a well-integrated LLVM-based JIT compiler, which is considered the industry standard. Additionally, PostgreSQL provides a simple API for integrating alternative JIT compiler. The scope of JIT compilation required for the TPC-H [14] benchmark is minimal: only two functions need to be compiled – tuple deforming and expression evaluation. Notably, expression evaluation does not require support for all operation types to handle TPC-H benchmark completely.

2. Taxonomy of JIT compilers

JIT compilers can be broadly classified into three categories based on their design philosophy, optimization capabilities and compilation overhead: heavyweight, medium-weight and lightweight.

Heavyweight JIT compilers are characterized by their extensive intermediate representations and long machine-independent optimization pipeline. Examples include LLVM and GCC [15], which are originally designed as optimizing AOT (Ahead-of-Time) compilers but can also be used as JIT compilers. This category also includes JIT compilers tightly coupled with specific run-time environments, tailored to particular object models, garbage collectors or bytecode peculiarities. Notable examples include V8 [16], C1/C2 (HotSpot) [17], RyuJIT (.NET) [18], Parrot (Perl) [19] and MoarVM (Raku) [20]. While these compilers typically generate highly optimized code, they either incur significant compilation latency or are difficult/impossible to integrate into systems outside their native run-time environments.

Medium-weight JIT compilers use a lightweight intermediate representation with reduced set of optimizations. Following the 80/20 principle, they aim to deliver 80% of GCC -O2 performance level with only 20% of the code. Due to their resemblance to scaled-down versions of LLVM, these compilers are sometimes referred to as “mini-LLVM”. A notable example is GNU LibJIT [21].

Lightweight JIT compilers lack a high-level IR and perform no machine-independent optimizations. At most, they may use a very low-level IR, effectively acting as platform-independent assemblers. These compilers prioritize fast code generation and emit machine code with minimal overhead. Examples include DynASM [22], AsmJIT [23], GNU lighting [24], sljit [25], xbyak [26], etc.

3. Lightweight JIT compilers

In the current phase of the research, we explored two lightweight JIT compilers: AsmJIT and DynASM. AsmJIT is a mature project with industrial applications that offers an abstraction with automatic register allocation – an interesting feature for comparison against manual approach in the same task. DynASM, on the other hand, promises outstanding code generation performance due to its preprocessing step and is part of widely used LuaJIT compiler for the Lua programming language.

3.1 DynASM

DynASM is a dynamic assembler designed for code generation engines and was originally developed as part of the LuaJIT project. It was initially created to serve as x86 JIT compiler in LuaJIT 1.1 (last released in 2010). In the current LuaJIT 2.1 version, DynASM is primarily used to generate the interpreter core, which is written in hand-optimized assembly. LuaJIT 2.0 introduced a new trace-based JIT compiler [27] with SSA-based intermediate representation along with numerous optimizations. This version also replaced DynASM with a new machine code generator for better performance in the JIT compiler. DynASM preprocesses source file with a mixture of C/C++ and assembly code and produces modified output file. The assembler directives are replaced with runtime calls, primarily “dasm_put(...)”, while the corresponding assembly instructions are embedded in an “actionlist” array in internal format. This array contains both the raw instructions and opcode-like metadata that defines how the code should be dynamically patched with actual values during runtime code generation phase. This micro-template-based approach considerably reduces code emission overhead. DynASM’s minimal runtime library handles code generation,

relocation and linking by processing the precompiled “actionlist”. As LuaJIT 2.1 maintains x86, x64, ARM, ARM64, PPC and MIPS architecture, DynASM correspondingly supports these platforms to generate the assembly-written interpreter core for each target architecture.

3.2 AsmJIT

AsmJIT is a lightweight, high-performance C++ library for dynamic code generation, supporting x86, x64, AArch64 architectures, with AArch32 support currently in development. Its well-thought architecture offers multiple multiple layers of abstraction, each providing varying levels of manual control to suit different use cases:

1. Assembler – The lowest-level interface, offering minimal abstraction over emitted instructions. It directly emits machine code into code buffer with no further transformations.
2. Builder – Built on top of Assembler, this layer provides simple code representation, which wraps each instruction into a node within a double-linked list. This structure enables flexible instruction rearrangement and basic code analysis, i.e. register usage tracking. The latter can be useful to optimize controlled call sites, reducing register pressure by neglecting strict calling convention to some extent.
3. Compiler – The highest-level abstraction, extending the Builder interface with virtual registers, automated register allocation, and ABI compliance for function calls. For register allocation it uses linear-scan algorithm with live-range analysis, simplifying a code generator implementation.

Layered design enables applications to combine different emitters as needed – manual register management for hot paths and automatic allocation for less critical code. AsmJIT is a well-established project that has high recognition in academic and several industrial level applications, e.g., Erlang JIT compiler [28].

4. Evaluation

4.1 Scope of JIT compilation

For evaluation purposes, we have selected PostgreSQL 17.2 as a candidate for experimentation. This DBMS follows a classical query processing pipeline (Fig. 1) that starts with lexical and syntactic analysis to parse client input and produce a parse tree. The parse tree then undergoes semantic analysis, where it is augmented with system and schema metadata to produce a query tree. Subsequently, a series of rewrite rules transform and canonicalize the query tree. Finally, planning and optimization convert the query tree into the most cost-effective logical plan, which is then translated into a physical plan suitable for execution via interpretation.

The PostgreSQL interpreter can be roughly divided into two parts. The first one implements data access, join operations and auxiliary operators such as Limit, Order by, Union (All). The second handles expression evaluation. These two parts are tightly integrated. For instance, during most join operations, the expression engine is invoked to compare attribute values between tuples. The expression engine uses a bytecode-like approach (Fig. 2), where the selected version implements 100 distinct opcodes, each representing a specific operation. Instruction flow between opcodes is managed via a dispatch table – provided the compiler used to build the DBMS binary supports computed gotos. This table contains addresses pointing to the relevant code block for each opcode. If computed gotos are unsupported, a standard switch statement is used as a fallback mechanism.

To achieve full compilation support for all expressions in the TPC-H benchmark, code generation must be implemented for a subset of 35 opcodes. Among these, one of the most performance-critical operations is tuple deforming, which converts on-disk tuple representation into in-memory format. This operation can be accelerated by creating a function specific to the table layout and the set of

columns being accessed. To ensure a proper comparison across different JIT implementations, all alternative providers must support compilation of this defined subset of expression engine, including the tuple deforming operation.

To implement code generation for required opcodes, the JIT compiler only needs to support a minimal set of operations – basic arithmetic and bitwise operations (mainly for computing offsets), memory loads/stores, branch instructions with labels (to implement control flow), and label referencing (to efficiently lower switch statements into jump tables).

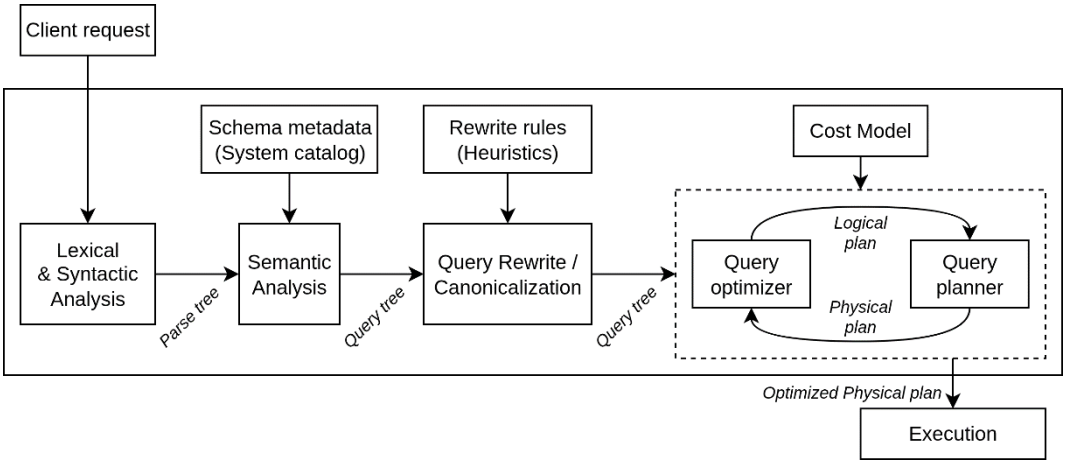


Fig 1. Classical query processing pipeline in DBMS.

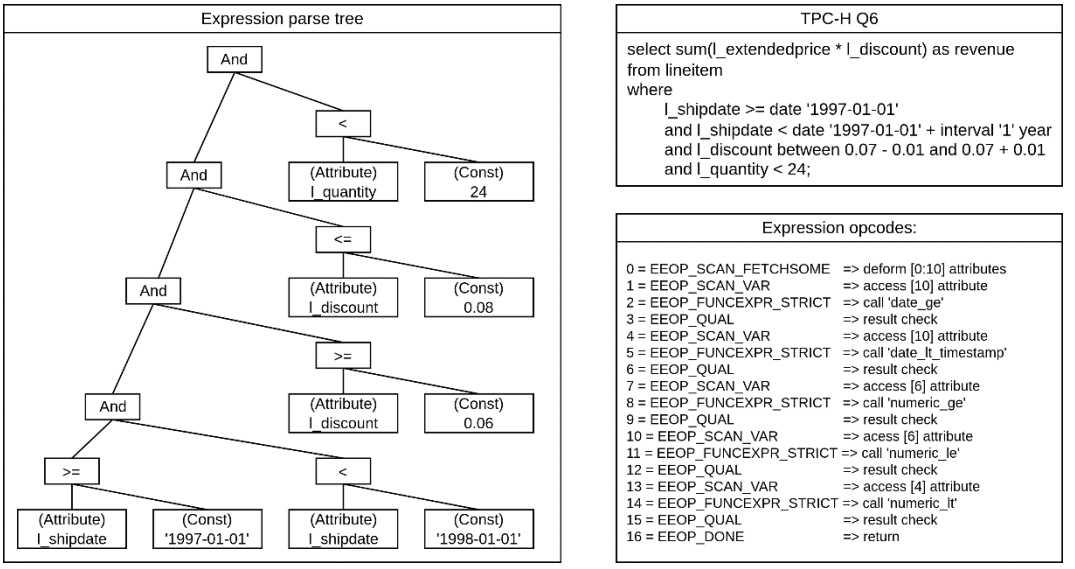


Fig 2. PostgreSQL expression opcodes for TPC-H Q6 WHERE clause.

4.2 Implementation fragment

We present a minimal excerpt from the implementation of lightweight JIT compilers (Fig. 3). In the case of AsmJIT, additional code wrappers are used to simplify field offset calculations and ensure that load and store operations match the specified data types. DynASM, on the other hand, uses preprocessor-level directives for type declarations, which automatically generate the required field

offsets. Overall, the process of writing JIT-compiled code is no more complex than writing traditional assembly by hand – and in the case of AsmJIT, it is even simpler due to its higher-level abstraction that automates register allocation.

```
bool asmjit_compile_expr(struct ExprState *state) {
  // ...
  for (int opno = 0; opno < state->steps_len; opno++) {
    // ...
    switch (opcode) {
    // ... MORE OPCODES
    case EEOP_INNER_VAR:
    case EEOP_OUTER_VAR:
    case EEOP_SCAN_VAR: {
      x86::Gp v_slot = opcode == EEOP_INNER_VAR   ? v_innerslot
                     : opcode == EEOP_OUTER_VAR ? v_outerslot
                                                : v_scanslot;
      const auto attnum = op->d.var.attnum;
      x86::Gp v_resvalue = cc::load_const_voidptr(
        jcc, "v_resvalue[attnum]", op->resvalue);
      x86::Gp v_values = ccTupleTableSlot::load_tts_values(jcc, v_slot);
      x86::Gp v_value = cc::load_array_at<Datum>(jcc, v_values, attnum);
      cc::store_at(jcc, v_resvalue, v_value);

      x86::Gp v_resnull = cc::load_const_voidptr(
        jcc, "v_resnull[attnum]", op->resnull);
      x86::Gp v_nulls = ccTupleTableSlot::load_tts_isnull(jcc, v_slot);
      x86::Gp v_isnull = cc::load_array_at<bool>(jcc, v_nulls, attnum);
      cc::store_at(jcc, v_resnull, v_isnull);
      break;
    }
    }
  }
  // ...
}
```

```
bool dynasm_compile_expr(ExprState *state) {
  // ...
  for (int opno = 0; opno < state->steps_len; opno++) {
    // ...
    switch (opcode) {
    // ... MORE OPCODES
    case EEOP_INNER_VAR:
    case EEOP_OUTER_VAR:
    case EEOP_SCAN_VAR: {
      const int slot_register =
        opcode == EEOP_INNER_VAR    ? R_INNERSLOT
        : opcode == EEOP_OUTER_VAR ? R_OUTERSLOT
                                   : R_SCANSLOT;
      | mov64 T1, (uintptr_t) op->resvalue
      | mov T0, t_slot:v_slot->tts_values
      | mov T0, [T0+DATUM_OFF(op->d.var.attnum)]
      | mov [T1], T0
      |
      | mov64 T2, (uintptr_t) op->resnull
      | mov T0, t_slot:v_slot->tts_isnull
      | mov T0b, [T0+BOOL_OFF(op->d.var.attnum)]
      | mov [T2], T0b
      ;
      break;
    }
    }
  }
  // ...
}
```

Fig 3. Excerpt from the implementation of lightweight JIT compilers.

4.3 Performance evaluation of JIT compilers

To evaluate performance of JIT compilers in PostgreSQL, we generated a set of TPC-H databases at varying factors (Fig. 4), starting from extremely low scale factor (SF) = 0.01 up to SF = 20 – ensuring that the entire database could fit withing the Linux page cache of the testing system. These varying scale factors allow us to examine how different levels of code generation and optimization impact query execution performance across different dataset sizes.

Scale factor (SF)	0.01	0.1	0.5	1	2.5	5	7.5	10	15	20
Database Size	109M	372M	1.7G	3.2G	6.2G	20G	24G	29G	40G	50G

Fig 4. TPC-H benchmark database size by scale factor.

The JIT capabilities in PostgreSQL are controlled through several configuration flags, which vary depending on the underlying engine: LLVM, AsmJIT, or DynASM. For testing, we selected the following configurations that control different aspects of code generation and optimization:

- {engine}_E – code generation only for expression nodes without optimizations.
- {engine}_EO – code generation for expression nodes with optimizations applied.
- {engine}_ED – code generation for expression nodes and deform functions without optimizations.
- {engine}_EDO – extends ED with optimizations applied after code generation.

- {engine}_EDOI – enhances EDO with inlining of built-in functions' source code (e.g. date_ge, int4mul, int8div) into the generated module, enabling direct inlining during optimization rather than indirect calls via absolute address constants.

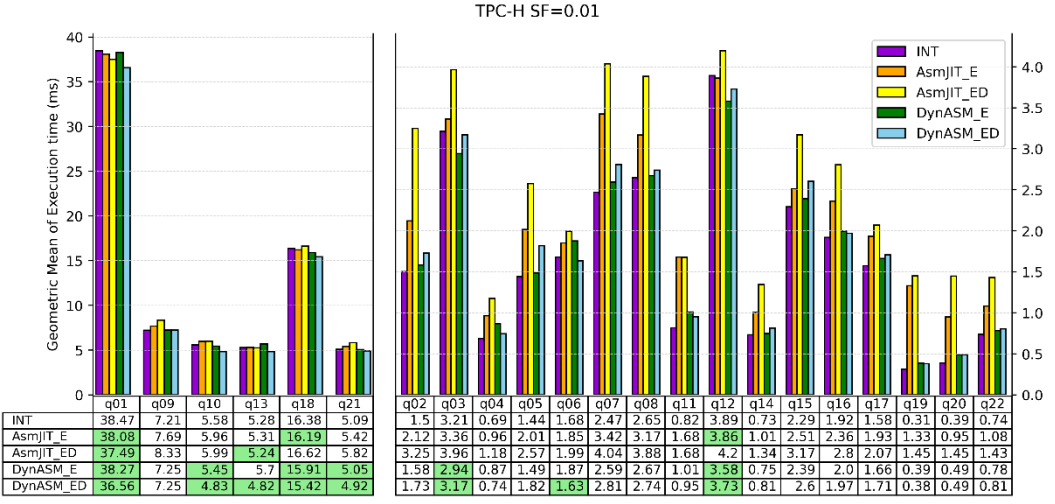
LLVM engine supports all five configurations, while AsmJIT and DynASM only use two – "{engine}_E" and "{engine}_ED" due to the absence of automatic optimizations and function inlining. Adding the latter would mean writing assembly generators by hand for each built-in function – a task that, while technically feasible, demands significant development effort.

All benchmarks were conducted on an Ubuntu 24.10 machine with an Intel i7-11700 CPU and 64GB DDR4 RAM (3200MHz). PostgreSQL 17.2 was built with GCC 14.2.0 and LLVM 18.1.8. PostgreSQL was configured with *shared_buffers* set to 8GB, *work_mem* to 256MB, and *max_parallel_workers_per_gather* to 0 to disable parallel query execution. Before each scale factor run, the entire database was preloaded into Linux page cache to eliminate I/O overhead. Each query was executed 21 times, with the first warmup run excluded; the geometric mean of execution times was recorded. Unless otherwise noted, execution times include all stages of query processing – from parsing to execution, including JIT compilation.

The results are presented in two parts. First, we compare the performance of AsmJIT and DynASM, focusing on both their compilation times and query execution times. Next, we evaluate the lightweight JIT engines against LLVM, using the interpreter as a baseline to assess the overall effectiveness of JIT compilation. We also analyze how the query execution performance of different JIT engines varies across database sizes and attempt to identify the threshold at which heavyweight JIT compiler begins to outperform the lightweight alternatives.

4.3.1 AsmJIT vs DynASM

Before proceeding, it is important to note that our AsmJIT-based JIT implementation relies on its *Compiler* abstraction layer, which automatically manages both register allocation and function calls. While this abstraction simplifies development, it leads to less predictable code quality and increased compilation overhead. In contrast, DynASM-based implementation requires everything to be handled manually, which improves compilation time due to the absence of abstraction, but makes performance highly dependent on the effectiveness of manual register allocation.



	q01	q09	q10	q13	q18	q21	q02	q03	q04	q05	q06	q07	q08	q11	q12	q14	q15	q16	q17	q19	q20	q22
INT	38.47	7.21	5.58	5.28	16.38	5.09	1.5	3.21	0.69	1.44	1.68	2.47	2.65	0.82	3.89	0.73	2.29	1.92	1.58	0.31	0.39	0.74
AsmJIT_E	38.08	7.69	5.96	5.31	16.19	5.42	2.12	3.36	0.96	2.01	1.85	3.42	3.17	1.68	3.86	1.01	2.51	2.36	1.93	1.33	0.95	1.08
AsmJIT_ED	37.49	8.33	5.99	5.24	16.62	5.82	3.25	3.96	1.18	2.57	1.99	4.04	3.88	1.68	4.2	1.34	3.17	2.8	2.07	1.45	1.45	1.43
DynASM_E	38.27	7.25	5.45	5.7	15.91	5.05	1.58	2.94	0.87	1.49	1.87	2.59	2.67	1.01	3.58	0.75	2.39	2.0	1.66	0.39	0.49	0.78
DynASM_ED	36.56	7.25	4.83	4.82	15.42	4.92	1.73	3.17	0.74	1.82	1.63	2.81	2.74	0.95	3.73	0.81	2.6	1.97	1.71	0.38	0.49	0.81

Fig 5. Geometric mean of execution time of lightweight JIT engines on TPC-H SF=0.01. Green table cells indicate that the JIT query execution time is lower than that of the interpreter.

In the context of TPC-H benchmark with a very small scale factor = 0.01, the execution time comparison between AsmJIT and DynASM (as shown in Fig. 5) reveals several observations:

- DynASM_E configuration outperforms AsmJIT_E across all queries except for q6 and q13. We suspect that the difference in performance for q1 and q6 may be due to execution-time fluctuations, as the median execution time is lower (Fig. 6). For q13, DynASM_E is consistently slower than AsmJIT_E, likely due to lower-quality code – a subject for future analysis.
- DynASM_ED configuration consistently outperforms AsmJIT_ED across all queries and AsmJIT_E, except for q15.
- ASMJIT_E improves performance in queries q1, q12, and q18, while ASMJIT_ED shows improvement in q1 and q13.
- DynASM_E achieves performance gains in queries q1, q3, q10, q12, q18, and q21, whereas DynASM_ED demonstrates improvements across a broader set of queries: q1, q3, q6, q10, q12, q13, q18, and q21.
- When considering the geomean speedup against the interpreter across all queries – AsmJIT_E: 0.758x, AsmJIT_ED: 0.653x, DynASM_E: 0.947x, and DynASM_ED: 0.949x – it becomes evident that compiling the deform function using AsmJIT actually degrades performance compared to the interpreter. In contrast, DynASM_ED slightly improves overall performance compared to the DynASM_E configuration. Notably, even at this minimal scale factor, DynASM achieves nearly neutral performance degradation, which we attribute to its highly efficient code generation and emission process.

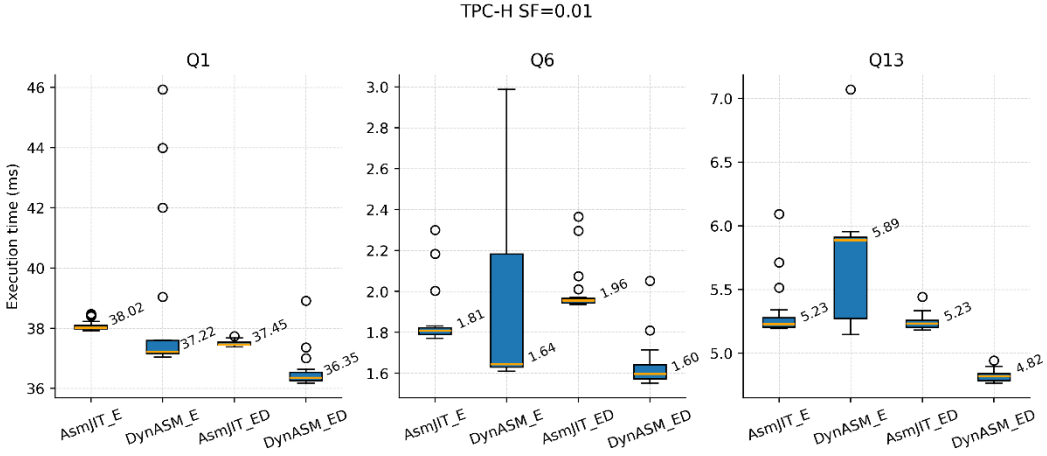


Fig. 6. Distribution of execution time of lightweight JIT engines on TPC-H SF=0.01 queries q1, q6, q13. The value near each box indicates the median execution time across 20 query runs.

Regarding compilation time (Fig. 7), that remains constant across all scale factors, we observe:

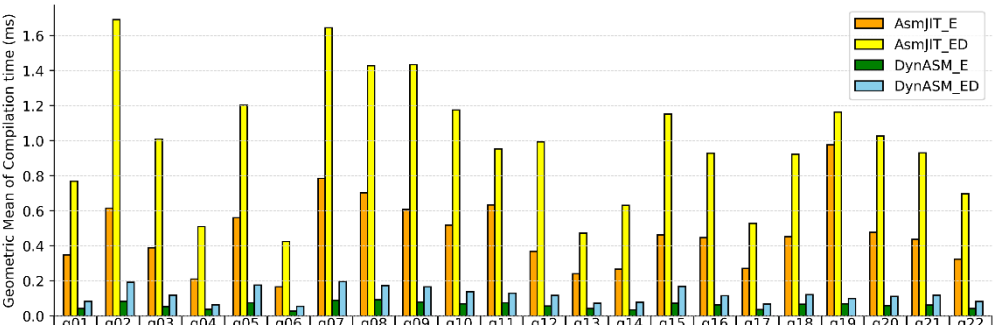
- DynASM_E achieves a geometric mean of compilation speedup of 7.5x over AsmJIT_E.
- DynASM_ED achieves a geometric mean of compilation speedup of 8.2x over AsmJIT_ED.

This shows that the DynASM-based implementation has significantly faster code generation and emission steps compared to AsmJIT-based implementation using its *Compiler* abstraction.

Finally, using AsmJIT_E as the baseline for code quality evaluation (Fig. 8), we observe the following:

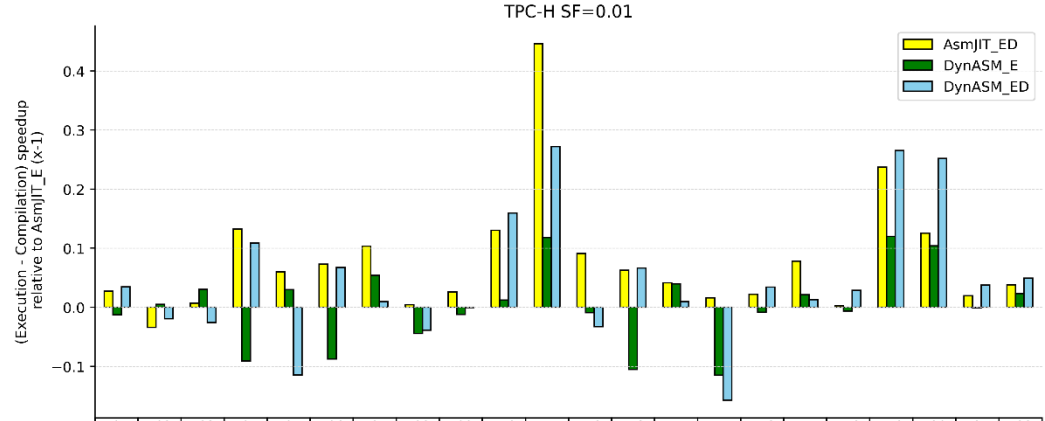
- DynASM_E produces lower-quality code in q1, q4, q6, q8, q9, q10, q12, q13, q15, q16, q18 and q21, with a peak regression of 0.88x measured in q15. Conversely, the maximum performance improvement is a 1.12x speedup, recorded in q19.

- The geometric mean speedup of DynASM_E is close to 1x, indicating that its overall code quality is comparable to AsmJIT_E.
- DynASM_ED exhibits a geometric mean speedup of 1.041x, while AsmJIT_ED achieves 1.074x – indicating that AsmJIT generates noticeably better code for the deform function, despite the hand-tuned assembly in the DynASM implementation, which avoids register spills entirely within the body of the function, with only minor spillage occurring in the prologue and epilogue.



	q01	q02	q03	q04	q05	q06	q07	q08	q09	q10	q11	q12	q13	q14	q15	q16	q17	q18	q19	q20	q21	q22
AsmJIT_E	0.35	0.61	0.39	0.21	0.56	0.17	0.78	0.7	0.61	0.52	0.63	0.37	0.24	0.27	0.46	0.45	0.27	0.45	0.98	0.48	0.44	0.32
AsmJIT_ED	0.77	1.69	1.01	0.51	1.2	0.42	1.64	1.43	1.43	1.17	0.95	0.99	0.47	0.63	1.15	0.93	0.53	0.92	1.16	1.03	0.93	0.7
DynASM_E	0.04	0.08	0.05	0.04	0.07	0.03	0.09	0.09	0.08	0.07	0.07	0.06	0.04	0.03	0.07	0.06	0.03	0.07	0.07	0.06	0.06	0.04
DynASM_ED	0.08	0.19	0.12	0.06	0.18	0.05	0.19	0.17	0.17	0.14	0.13	0.12	0.07	0.08	0.17	0.12	0.07	0.12	0.1	0.11	0.12	0.08

Fig 7. Compilation time of Lightweight JIT engines on TPC-H benchmark.



	q01	q02	q03	q04	q05	q06	q07	q08	q09	q10	q11	q12	q13	q14	q15	q16	q17	q18	q19	q20	q21	q22
AsmJIT_E	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
AsmJIT_ED	0.03	-0.03	0.01	0.13	0.06	0.07	0.1	0.0	0.03	0.13	0.45	0.09	0.06	0.04	0.02	0.02	0.08	0.0	0.24	0.13	0.02	0.04
DynASM_E	-0.01	0.0	0.03	-0.09	0.03	-0.09	0.05	-0.04	-0.01	0.01	0.12	-0.01	-0.1	0.04	-0.12	-0.01	0.02	-0.01	0.12	0.1	-0.0	0.02
DynASM_ED	0.03	-0.02	-0.03	0.11	-0.11	0.07	0.01	-0.04	-0.0	0.16	0.27	-0.03	0.07	0.01	-0.16	0.03	0.01	0.03	0.27	0.25	0.04	0.05

Fig 8. Execution - Compilation time speedup, relative to AsmJIT_E. The metric is defined as (Speedup - 1), where Speedup = (AsmJIT_E["Execution time"] – AsmJIT_E["Compilation time"]) / (Target ["Execution time"] - Target["Compilation time"]).
Both execution and compilation times represent geometric mean values based on 20 query runs.
Value of 0 corresponds to equal performance;
positive values denote speedups, and negative values denote slowdowns.

4.3.2 Lightweight against LLVM

We begin our comparison of lightweight JIT engines against LLVM using the compilation time heat map shown in Fig. 9, which graphically summarizes the observed patterns. The values represent the percentage of total query execution time that is spent on compilation, across various JIT engine

configurations and scale factors that increase by an order of magnitude. As the scale factor increases, the relative compilation overhead decreases, indicating that compiled execution begins to outperform interpreter-based execution for an increasing number of queries (denoted by the ▲ symbol in the figure). In the case of LLVM, its various JIT configurations only begin to justify their compilation overhead starting at SF=1, with the notable exception of query q1 at SF=0.1. Query q1 has the longest execution time in the benchmark, which is reflected in its relatively low compilation time percentage compared to other queries. Furthermore, at SF=1, both lightweight JIT compilers already demonstrate improved execution time over the interpreter across most queries. In contrast, LLVM – even in its most lightweight configuration LLVM_E – does not consistently outperform the interpreter, indicating that its compilation overhead remains significantly higher.

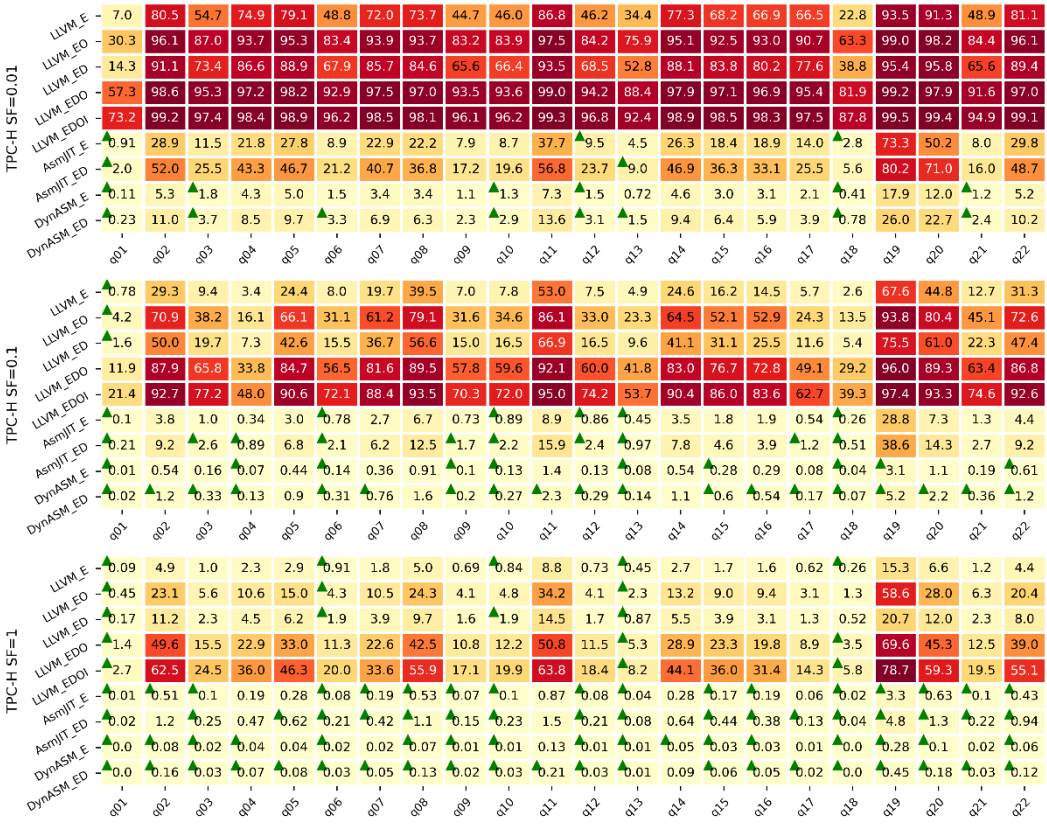
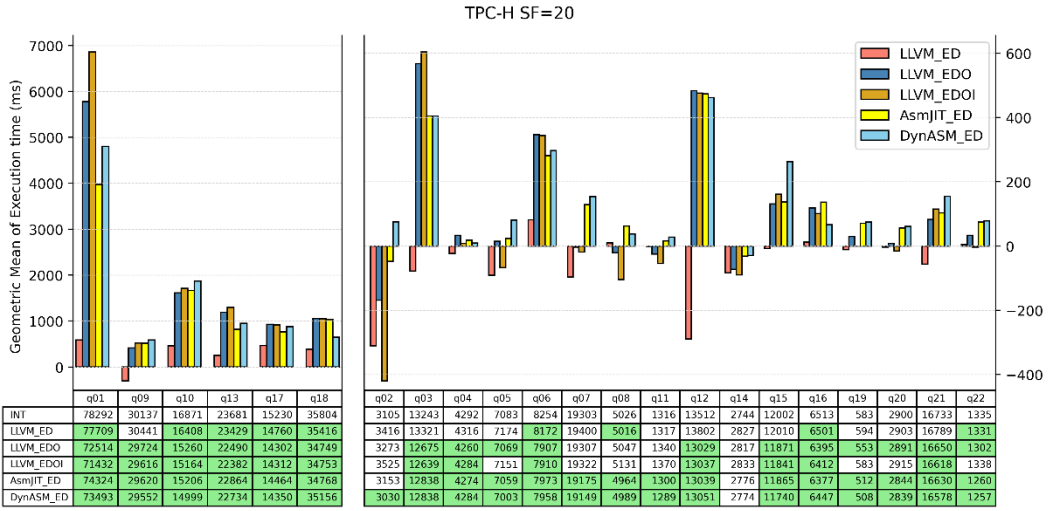


Fig 9. Compilation time as a percentage of total query execution time across TPC-H benchmark scale factors for various JIT engine configurations. Both execution and compilation times represent geometric mean values based on 20 query runs. ▲ indicates that a particular configuration achieves better performance than the interpreter for a given query.

Next, we compare only the most compilation-heavy JIT configurations on the largest of scale factors in our tests, SF=20 (Fig. 10), to examine performance behavior on a relatively large dataset. Interpreter execution time served as the baseline: positive values indicate that JIT execution is faster than interpretation by the corresponding number of milliseconds, while negative indicate it is slower by that amount. The results show that only one query – q14 – does not benefit from any JIT engine. DynASM_ED consistently improves performance across all queries except q14. Similarly, AsmJIT_ED also shows improvements across the board, with an additional exception of q2. LLVM_EDO improves performance on 17 out of 22 queries, achieving the best absolute per-query performance on q4, q6, q12, q17, and q18. Meanwhile, LLVM_EDOI shows the best absolute per-

query performance only on q1, q3, and q13. This suggests that further increasing the scale factor would likely lead to broader performance benefits across all queries when using the LLVM JIT with full optimization enabled, despite its higher compilation overhead.

Interestingly, Q1 – the most time-consuming query – shows a relatively small performance gap between the lightweight JITs and LLVM. For this query LLVM_EDOI incurs a total compilation time of approximately 100ms, whereas DynASM_ED requires less than 0.1 ms. When calculating the relative performance improvements – DynASM_ED ~6.13%, LLVM_EDOI ~8.76%, and idealized "zero-compile-time" LLVM_EDOI ~8.88% – we observe that DynASM_ED achieves approximately two-thirds of the performance gain offered by LLVM on this long-running query. This indicates that, at relatively high scale factors, lightweight JITs can maintain competitive performance compared to LLVM.



	q01	q09	q10	q13	q17	q18
INT	78292	30137	16871	23681	15230	35804
LLVM_ED	77709	30441	16408	23429	14760	35416
LLVM_EDO	72514	29724	15260	22490	14302	34749
LLVM_EDOI	71432	29616	15164	22382	14312	34753
AsmJIT_ED	74324	29620	15206	22864	14464	34768
DynASM_ED	73493	29552	14999	22734	14350	35156

	q02	q03	q04	q05	q06	q07	q08	q11	q12	q14	q15	q16	q19	q20	q21	q22
INT	3105	13243	4292	7083	8254	19303	5026	1316	13512	2744	12002	6513	583	2900	16733	1335
LLVM_ED	3416	13321	4316	7174	8172	19400	5016	1317	13802	2827	12010	6501	594	2903	16789	1331
LLVM_EDO	3273	12675	4260	7069	7907	19307	5047	1340	13029	2817	11871	6395	553	2891	16650	1302
LLVM_EDOI	3525	12639	4284	7151	7910	19322	5131	1370	13037	2833	11841	6412	583	2915	16618	1338
AsmJIT_ED	3153	12838	4274	7059	7973	19175	4964	1300	13039	2776	11865	6377	512	2844	16630	1260
DynASM_ED	3030	12838	4284	7003	7958	19149	4989	1289	13051	2774	11740	6447	508	2839	16578	1257

Fig 10. Execution time difference: (Interpreter - JIT) on TPC-H SF=20. Green table cells indicate that the JIT query execution time is lower than that of the interpreter.

Next, we take a closer look at run-time performance and compilation characteristics of the JIT configurations shown in Figure 11. First, there is a significant gap in compilation time between lightweight JIT engines and LLVM-based configurations. The difference between the basic configurations of DynASM_E and LLVM_E is 69.4x. When optimizations are enabled in LLVM (LLVM_EO), the gap becomes even more pronounced – 397.9x. Second, the difference in compilation time between the lightweight configurations of DynASM and AsmJIT is 6.8x and 7.75x respectively. We attribute this primarily to AsmJIT's automatic register allocation, which introduces additional overhead compared to DynASM. Regarding run-time performance, we observe that all JIT configurations generate more efficient machine code than the interpreter. The best run-time result is achieved by the LLVM_EDOI configuration, completing execution of the entire benchmark in less than 12 seconds, excluding compilation time. Lightweight configurations with deform function generation (*_ED) show performance close to that of the optimized LLVM configuration (LLVM_EDO). However, when both run-time and compilation time are considered together, LLVM-based configurations lose their run-time advantage. For example, the LLVM_EDOI configuration, which has the best run-time performance, reaches a total execution time of 14.49 seconds when compilation time is included. In contrast, lightweight JIT engines, which has almost no compilation overhead, achieve noticeably lower total execution times.

Finally, we compare the overall performance of the JIT engines across all scale factors (Fig. 12) to determine whether there is a threshold at which the lightweight JIT engines begin to lose ground to LLVM as the scale factor increases.

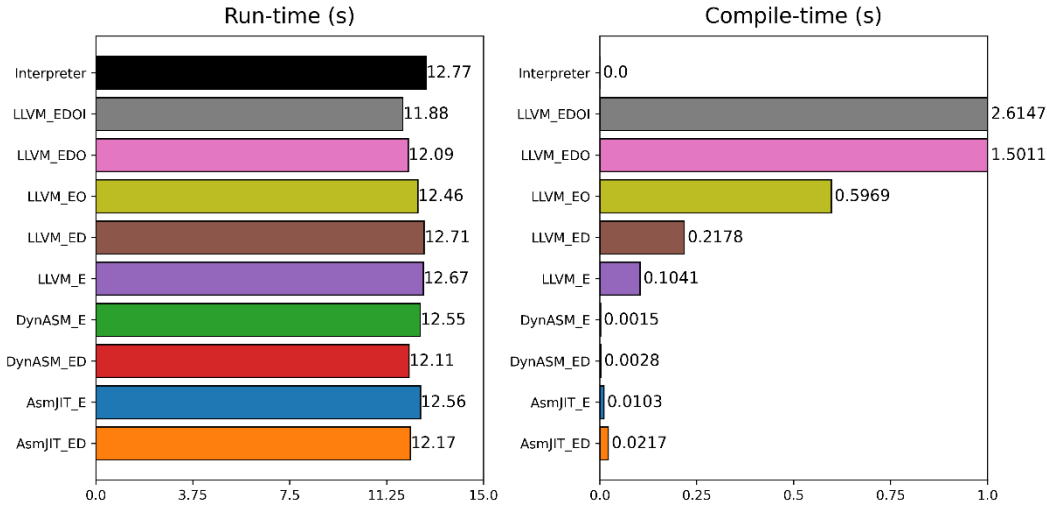


Fig 11. Compile-time and run-time accumulated over all TPC-H queries at scale factor 1. Each query executed 21 times; 1st warm-up run is discarded and geometric mean is computed over the remaining runs.

The results calculated using geometric mean show that when overall system throughput is prioritized over peak per-query performance, such a shift is not clearly evident. Across all scale factors, DynASM_ED maintains the leading position, thanks to its fast code generation and emission speed – even though it generates lower-quality code compared to AsmJIT_ED. AsmJIT secures second place starting from SF=0.5, but ranks third on the two smallest scale factors, where it loses the second position to DynASM due to slower code generation. LLVM_EDO only reaches third place starting at SF=15, while LLVM_EDOI begins to show overall improvement starting at SF=20, suggesting that higher scale factors are necessary to fully benefit from LLVM's optimization pipeline.

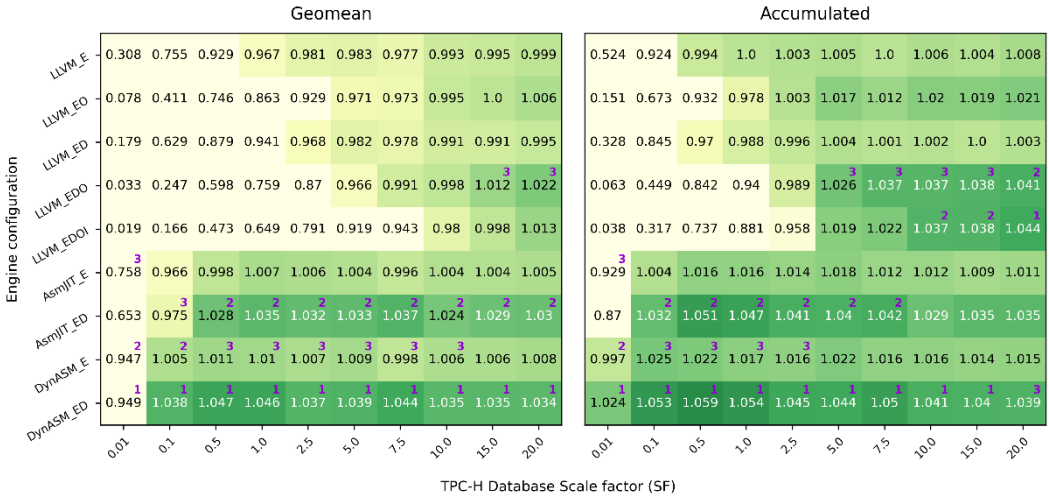


Fig 12. JIT speedups for all queries across all tested scale factors relative to the interpreter, shown in two ways: (left) geometric mean of execution times, and (right) total sum of execution time. The numbers in the top right corner of the cells show the top 3 results ranked by improvement at each scale factor. Cell values indicate the speedup (or slowdown) relative to the interpreter, expressed as × times.

The results, calculated as a sum of execution time of all queries, show that starting from scale factor 10, LLVM ranks second and third, with only DynASM outperforming it. By scale factor 20, LLVM surpasses DynASM and claims the top two positions. This can be explained by the varying execution times of TPC-H queries. They can be roughly grouped into two categories: short-running and long-running queries. The latter start to take significantly longer time to execute as the scale factor increases and benefit more from high-quality machine code, even at lower scale factors. For example, at scale factor 20, query 9 takes around 30 seconds to complete, while query 2 finishes in just 3 seconds – a tenfold difference. Because of this imbalance, the long-running queries make up most of the total benchmark execution time. This gives LLVM a performance advantage over lightweight JIT engines, as improvements in these costly queries have a bigger effect on the total runtime. LLVM's advantage is expected to grow with higher scale factors, since compilation time becomes less significant.

5. Conclusion

In this paper, we evaluated two lightweight JIT compilers – DynASM and AsmJIT – against LLVM, using PostgreSQL as a common experimental environment. Our findings show the AsmJIT-based implementation, which utilizes a higher-level abstraction for automatic register allocation, produces higher-quality code compared to the DynASM-based implementation, which relies on manual register allocation. Nevertheless, DynASM's exceptional code generation speed allows it to outperform AsmJIT across all tested scales.

Both lightweight JIT compilers begin to improve system performance starting from a scale factor (SF) of 0.5, demonstrating their suitability even for very small queries. The query planner's cost model can be used to configure such JIT compilers to operate at low cost thresholds, without being overly sensitive to inevitable cost estimation inaccuracies.

While LLVM demonstrates better peak performance on certain queries, it generally lags behind lightweight JIT engines until reaching the largest tested scale factor (SF=20), where its performance gains become evident. This suggests that even larger scale factors are necessary for LLVM's extensive optimization pipeline to yield meaningful benefits.

As part of our future work, we plan to explore an AsmJIT-based implementation without compiler abstraction, in order to enable a more direct comparison of code generation and emission speed with DynASM. Additionally, we aim to investigate platform-independent JIT assembler alternatives, as our current implementations are limited to x86-64 architecture, and porting and maintaining them on other architecture requires considerable effort. We also intend to evaluate medium-weight JIT engines – often referred to as "mini-LLVM" style compilers – equipped with a minimal set of essential optimizations, again using PostgreSQL as our testbed.

References

[1]. PostgreSQL – open Source DBMS. [Online], Available at: https://www.postgresql.org, accessed 05.05.2025.

[2]. G. Graefe, "Volcano - An Extensible and Parallel Query Evaluation System," IEEE Trans. Knowl. Data Eng., vol. 6, no. 1, pp. 120-135, 1994. DOI: 10.1109/69.273032. [Online], Available at: https://doi.org/10.1109/69.273032.

[3]. T. Neumann, "Efficiently Compiling Efficient Query Plans for Modem Hardware," PVLDB, vol. 4, no. 9, pp. 539-550, 2011. DOI: 10.14778/2002938.2002940. [Online], Available at: http://www.vldb.org/pvldb/vol4/p539-neumann.pdf.

[4]. PostgreSQL mailing lists, "[GSoC] Push-based query executor discussion." [Online], Available at: *https://www.postgresql.org/message-id/87shm1zfnz.fsf%40ispras.ru*, accessed 05.05.2025.

[5]. A. Shaikhha, M. Dashti and C. Koch, «Push vs. pull-based loop fusion in query engines», CoRR, abs/1610.09166, 2016 arXiv: 1610.09166. Available at: *https://arxiv.org/abs/1610.09166*.

[6]. PostgreSQL 10.0 Release Notes, [Online] Available at: *https://www.postgresql.org/docs/release/10.0*, accessed 05.05.2025.

[7]. GitHub, Mirror of the official PostgreSQL GIT repository, "Faster expression evaluation and targetlist projection." Commit SHA: b8d7f053c5c2bf2a7e8734fe3327f6a8bc711755 [Online], Available at: *https://github.com/postgres/postgres/commit/b8d7f053c5c2bf2a7e8734fe3327f6a8bc711755*, accessed 05.05.2025.
[8]. PGCon 2017. A. Freund Integrating Just In Time Compilation, [Online] Available at: *https://www.pgcon.org/2017/schedule/attachments/462_jit-pgcon-2017-05-25.pdf*, accessed 05.05.2025.
[9]. The LLVM Foundation, The LLVM Compiler Infrastructure. [Online] Available at: *https://llvm.org*, accessed 05.05.2025.
[10]. PGCon 2018. A. Freund A.The State of Postgres JIT – 2018 Edition, [Online] Available at: *https://anarazel.de/talks/2018-06-01-pgcon-state-of-jit/state-of-jit.pdf*, accessed 05.05.2025.
[11]. A. Engelke and T Schwarz. 2024. Compile-Time Analysis of Compiler Frameworks for Query Compilation. In Proceedings of the 2024 IEEE/ACM International Symposium on Code Generation and Optimization (CGO '24). IEEE Press, 233–244. DOI: *https://doi.org/10.1109/CGO57630.2024.10444856*.
[12]. A. Kohn, V. Leis, T. Neumann "Adaptive Execution of Compiled Queries", 2018 IEEE 34th International Conference on Data Engineering (ICDE), Paris, France, 2018, pp. 197-208, DOI: 10.1109/ICDE.2018.00027.
[13]. T. Kersten, V. Leis, T. Neumann "Tidy Tuples and Flying Start: fast compilation and fast execution of relational queries in Umbra" The VLDB Journal 30, 5 (Sep 2021), 883–905. [Online], Available at: *https://doi.org/10.1007/s00778-020-00643-4*.
[14]. Transaction Processing Performance Council, "TPC-H Benchmark." [Online]. Available at: *http://www.tpc.org/tpch*, accessed 05.05.2025.
[15]. GCC, the GNU Compiler Collection, [Online], Available at: *https://gcc.gnu.org/*, accessed 05.05.2025.
[16]. V8 JavaScript Engine, [Online], Available at: *https://v8.dev/*, accessed 05.05.2025.
[17]. OpenJDK, The HotSpot Group, [Online], Available at: *https://openjdk.org/groups/hotspot/*, accessed 05.05.2025.
[18]. DotNet, RyuJIT compiler overview, [Online], Available at: *https://github.com/dotnet/runtime/blob/main/docs/design/coreclr/jit/ryujit-overview.md*, accessed 05.05.2025.
[19]. Parrot Virtual Machine, [Online], Available at: *http://www.parrot.org/*, accessed 05.05.2025.
[20]. MoarVM, a VM for NQP and Rakudo, [Online], Available at: *https://www.moarvm.org/*, accessed 05.05.2025.
[21]. GNU LibJIT – backend for DotGNU Portable.NET JIT engine, [Online], Available at: *https://www.gnu.org/software/libjit*, accessed 05.05.2025.
[22]. DynASM – dynamic assembler for code generation engines. [Online], Available at: *https://luajit.org/dynasm.html*, accessed 05.05.2025.
[23]. AsmJIT – lightweight library for low-latency machine code generation. [Online], Available at: https://asmjit.com, accessed 05.05.2025.
[24]. GNU lightning – library that generates assembly language code at run-time. [Online], Available at: https://www.gnu.org/software/lightning/, accessed 05.05.2025.
[25]. sljit – a low-level, platform-independent JIT compiler. [Online], Available at: https://zherczeg.github.io/sljit/, accessed 05.05.2025.
[26]. Xbyak – a JIT assembler for x86/x64 architectures. [Online], Available at: *https://github.com/herumi/xbyak*, accessed 05.05.2025.
[27]. M. Pall, "LuaJIT 2.0 intellectual property disclosure and research opportunities", [Online], Available at: *http://lua-users.org/lists/lua-l/2009-11/msg00089.htm*l, accessed 05.05.2025.
[28]. Erlang/OTP documentation "BeamAsm, the Erlang JIT", [Online] Available at: *https://www.erlang.org/doc/apps/erts/beamasm.html*, accessed 05.05.2025.

Информация об авторах / Information about authors

Михаил Вячеславович ПАНТИЛИМОНОВ – научный сотрудник отдела компиляторных технологий ИСП РАН. Научные интересы: статический анализ, компиляторные технологии, СУБД.

Mikhail Vyacheslavovich PANTILIMONOV – researcher at Compiler Technology department of ISP RAS. Research interests: static analysis, compiler technologies, DBMS.

Рубен Артурович БУЧАЦКИЙ – кандидат технических наук, научный сотрудник отдела компиляторных технологий ИСП РАН. Научные интересы: статический анализ программ, компиляторные технологии, оптимизации.

Ruben Arturovich BUCHATSKIY – Cand. Sci. (Tech.), researcher at Compiler Technology department of ISP RAS. Research interests: static analysis, compiler technologies, optimizations.

Денис Владиславович ЗАВЕДЕЕВ – аспирант Института системного программирования им. В.П. Иванникова Российской академии наук. Сфера научных интересов: компиляторы, языковые виртуальные машины.

Denis Vladislavovich ZAVEDEEV – postgraduate student at Ivannikov Institute for System Programming of the Russian Academy of Sciences. Research interests: compilers, language virtual machines.