Maintainable, Low-Latency and High-Quality Query Compilation with MLIR and TPDE

Jonas Ladner
Technical University of Munich
Munich, Germany
jonas.ladner@tum.de

Lukas Döllerer*
CedarDB
Munich, Germany
lukas.doellerer@cedardb.com

Alexis Engelke*
Univ. Grenoble Alpes, Inria, CNRS, Grenoble INP, LIG
38000 Grenoble, France
alexis.engelke@inria.fr

Michael Jungmair
Technical University of Munich
Munich, Germany
jungmair@in.tum.de

ABSTRACT

Query compilation is an effective execution paradigm, particularly for complex queries, diverse workloads, and heterogeneous hardware. The MLIR compiler framework has recently been adopted to build compiling query engines due to it being specifically designed to lower the engineering effort while still generating efficient code. However, MLIR-based engines incur compilation latencies of 50–150 ms for typical analytical SQL queries, which dominate end-to-end latency on smaller datasets.

We address this with two contributions: First, we introduce a low-latency MLIR compilation backend based on TPDE that bypasses the LLVM lowering and compilation pipeline. Second, we propose techniques that reduce MLIR's intrinsic overhead. We integrate both into LingoDB, our MLIR-based query engine, and evaluate them on the TPC-H and TPC-DS benchmarks. The new compilation backend generates machine code up to 66x faster, even outperforming Hyper's compilation backend, and our techniques reduce MLIR's overhead by more than 2x. Together, this allows LingoDB to achieve lower end-to-end query latencies than Hyper and DuckDB, even at scale factor 1.

VLDB Workshop Reference Format:
Jonas Ladner, Lukas Döllerer, Alexis Engelke, and Michael Jungmair. Maintainable, Low-Latency and High-Quality Query Compilation with MLIR and TPDE. VLDB 2026 Workshop: 17th International Workshop on Accelerating Analytics and Data Management Systems Using Modern Processor and Storage Architectures (ADMS26).

1 INTRODUCTION

For high-performance analytical data processing, state-of-the-art database engines either rely on vectorized execution to amortize interpretation overhead [4], or query compilation to avoid interpretation overhead altogether [17, 24]. Compiling queries to efficient machine code is especially interesting for 1) complex queries and expressions [18], 2) combining relational query processing with other forms of computations [15], and 3) targeting heterogeneous hardware where vectorized execution can be challenging to implement [15]. Despite the great performance profile and the higher flexibility, query compilation comes with its own set of challenges that often lead to new systems discarding it from the beginning [2].

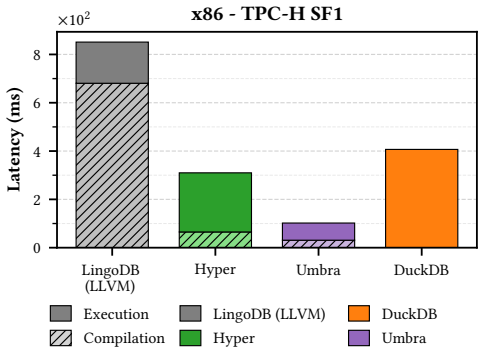


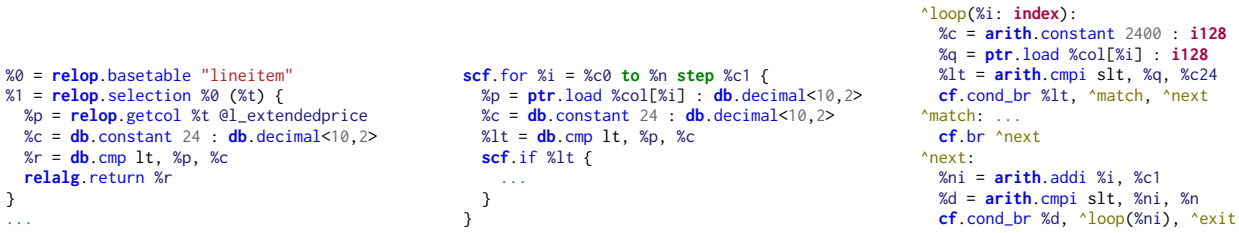
Figure 1: Compilation overhead dominates LingoDB's initial query latency on small datasets

Specifically, database architects typically face three connected, major challenges when building a compiling engine: (1) achieving low compilation latencies such that queries can quickly start processing tuples, (2) being able to generate highly performant code, and (3) keeping the engineering effort low, even when compiling database-specific operations and targeting multiple architectures [7, 19].

Prior approaches to query compilation typically address at most two of the three challenges. Systems that generate code in standard programming languages like C or C++ are comparatively simple to build, debug, and maintain, but introduce significant compilation latency due to slow, off-the-shelf compilers such as GCC or Clang [21]. On the other hand, query engines that emit an intermediate representation (IR) and directly use an existing compiler backend often still exhibit comparatively high compilation latencies (e.g., LLVM) and must manually translate custom operations into the backend's IR. For some of the more recent fast compilation backends, such as MIR [23], this translation can be especially challenging, e.g., due to the lack of 128-bit operations required to support decimals. Lastly, custom compiler backends pioneered by HyPer [17] and

*Work performed while at TUM.

Proceedings of the VLDB Endowment. ISSN 2150-8097.



(a) High-level Operators **(b) Mid-level structured control flow (`scf`)** **(c) Low-level `cf` + `arith` (basic blocks)**

Figure 2: Different possible representations for fragments of a SQL query in MLIR: (a) using database-specific, high-level operators, (b) structured control-flow and database-specific types, (c) lower-level MLIR dialects.

Umbra [24] achieve leading performance at the cost of technical complexity [7, 19].

In the past 5 years, MLIR has established itself as a framework for reducing the complexity and engineering effort of building domain-specific compilers [22], and has been quickly adopted for different use cases in data processing: LingoDB [16] uses MLIR for the full stack of query optimization and compilation, while Daphne [5] builds on MLIR only for high-level optimizations and abstractions, and NebulaStream builds on MLIR for easier query compilation [8]. As already discussed and evaluated in previous works [8, 13, 16], building on MLIR keeps engineering effort low, enables high flexibility and maintainability, and achieves high performance through the default LLVM-based backend.

However, previous work [8, 16] also found that building on MLIR yields compilation latencies in the range of 2-digit milliseconds, which dominate overall latencies for small datasets. For example, Figure 1 shows that LingoDB's overall end-to-end latencies are significantly higher than those of other compilation-based (Hyper, Umbra), and vectorized (DuckDB) systems for TPC-H scale factor 1, due to the high compilation overhead. More specifically, building on MLIR today introduces higher latencies for two reasons: (1) using LLVM as a general-purpose compilation backend not optimized for low-latency compilation, and (2) MLIR's flexible design increases the cost of transformation and optimization passes.

In this paper, we address both challenges systematically, ensuring that data systems built on MLIR are not only highly performant and maintainable but also competitive for small datasets. To address (1), we introduce a new *direct* compilation backend for MLIR based on TPDE [25] that avoids expensive preparation steps in MLIR and benefits from TPDE's design, which already aims for low-latency compilation and portability. Additionally, we address (2) by discussing several techniques for reducing the time spent in MLIR for high-level optimizations and transformations. We show the practical impact by integrating MLIR-TPDE, the new compilation backend, and the proposed techniques into LingoDB. Our evaluation shows that using the new compilation backend, we can compile a lower-level MLIR module similarly fast as custom compilation backends used by, e.g., Umbra and Hyper, while generating efficient code for both x86-64 and ARM. Furthermore, we show that the proposed techniques can significantly reduce MLIR's overhead and, in combination, allow us to achieve competitive end-to-end query latencies even for small datasets. Finally, we also discuss that the implementation effort for the new backend is small and that most of it can be reused by others.

2 BACKGROUND

In the following section, we first introduce the MLIR compiler framework and the recently introduced TPDE framework.

2.1 MLIR

MLIR [22] is a general-purpose compiler framework that aims to reduce the cost of developing domain-specific compilers by providing developers with modular and extensible compiler tooling. Its core abstraction is the operation, which represents a unit of computation. Operations, along with types and attributes, are grouped into dialects that define extensible namespaces that provide domain-specific abstractions. In contrast to existing, more low-level compiler frameworks like LLVM, MLIR also allows nested operations belonging to a parent operation.

More specifically, this nesting allows for representing domain-specific computations at different abstraction levels, as sketched in Figure 2 for query processing. Figure 2a sketches how declarative, relational operators can be modeled as an MLIR dialect, Figure 2b shows the same computation at a medium abstraction level with structured control-flow and concrete database-specific types and operations, and Figure 2c sketches the equivalent MLIR module using only basic blocks and fundamental MLIR operations. Basic blocks consist of a sequence of instructions, which only allow for branching at either the start or the end of the block. In the case of Figure 2c, we use MLIR operations from the `arith` (common arithmetic operations on integer and floats), `ptr` (memory loads and stores) and `cf` (control-flow between basic blocks using jumps) dialects.

The conversion between higher-level and lower-level representations is performed by applying sequences of MLIR passes that transform the IR while preserving semantics. A pass may rewrite operations within the same dialect or replace them with operations from a different, typically lower-level dialect. This progressive, multi-level lowering is one of MLIR's main strengths, as each transformation and optimization can operate at the abstraction level most natural for it. After a low-enough representation is reached (corresponding to Figure 2c), MLIR typically uses the LLVM ecosystem to just-in-time (JIT) generate machine code. This process consists

of four phases: First, the MLIR module is lowered to the MLIR-provided `llvm` dialect. Then, this MLIR `llvm` dialect is converted to LLVM-IR. After this conversion step, several optimization passes are run on the LLVM-IR. Lastly, the actual LLVM machine code generation is invoked.

2.2 TPDE

Recently, TPDE [25] was open-sourced as a framework to build low-latency compiler backends. One key distinction between TPDE and existing compiler backends is that it directly attaches to the existing IR without forcing a system to transform its IR to a compiler-specific IR. Instead, it requires an adapter to the existing IR, allowing TPDE to extract information and traverse it.

While TPDE itself is not a compiler backend, it minimizes the engineering burden by providing complex components like liveness analysis, register allocation, instruction encoding, and the implementation of calling conventions. Furthermore, it allows the user to specify the semantics of operations by writing snippets in C or C++, which are compiled at build time into efficient C++ functions (an encoder) that generate optimized assembly when invoked. TPDE achieves this by leveraging LLVM to turn those snippets into LLVM's machine IR, which is then inspected to generate the encoder. These generated encodings can then be called platform-agnostically, significantly reducing engineering effort by allowing the vast majority of the code to be written target-independently.

3 MLIR-TPDE COMPILATION BACKEND

As shown in Figure 1, and discussed in Section 2.1, the LLVM backend imposes high compilation latency by first requiring the conversion to LLVM-IR and then relying on the slow LLVM compiler infrastructure. In this section, we now introduce MLIR-TPDE, a novel TPDE-based compilation backend for MLIR. It supports built-in, lower-level MLIR dialects that are typically present in the MLIR module, just before the backend-specific lowering pipeline starts: `func`: functions and function calls, `cf`: control-flow between basic blocks in the form of (conditional) jumps, and `arith`: common arithmetic operations on integers and floats. Specifically, we do not aim to support higher-level dialects such as structured control flow, as this would require MLIR-TPDE to create virtual basic blocks, e.g., for loops, thereby increasing complexity significantly. In the remainder of this section, we describe the three distinct components of MLIR-TPDE: a target-independent IR-adapter that bridges MLIR and TPDE, target oblivious C snippets that define the semantics of MLIR operations, and finally, target-specific compiler backends. Furthermore, we discuss how database-specific, custom MLIR operations can be supported.

IR Adapter. As already described in Section 2.2, TPDE differs from existing compiler frameworks in that it does not prescribe an IR that needs to be used, but instead requires writing a TPDE adapter for an existing IR such that TPDE can walk over it. For MLIR, this required exposing the structure of the MLIR module in terms that TPDE can understand: functions, basic blocks of operations, and jumps connecting basic blocks. The main challenge here was to bridge two different static single assignment (SSA) paradigms: TPDE relies on ϕ-nodes, while MLIR relies on block arguments. A ϕ-node selects one of several incoming values depending on which predecessor edge control flow arrived from, since SSA form assigns each variable exactly once and a value reaching a block may differ per path. A loop variable, for example, takes its initial value along the entry edge and its updated value along each back-edge. MLIR-TPDE's IR Adapter emulates ϕ-nodes by explicitly looking up all predecessor blocks and analyzing the corresponding jump operations that provide the input arguments for the current block.

```
// snippet for adding two 32-bit unsigneds
unsigned addU32(unsigned a, unsigned b) {
 return a + b;
}
// snippet for adding two 64-bit unsigneds
unsigned long addU64(unsigned long a, unsigned long b) {
 return a + b;
}
```

(a) C snippets for `arith.addi`

```
bool compile_arith_addi(IRInstRef inst) {
 // prepare value references from op arguments
 auto lhs_ref = this->val_ref(inst->getOperand(0));
 auto rhs_ref = this->val_ref(inst->getOperand(1));

 // prepare result reference
 auto res_ref = this->result_ref(inst);

 auto bitwidth = inst->getResult(0).getType().getBitWidth();
 // call encoder generated from the snippets depending on bitwidth
 switch(bitwidth){
    case 32:
      return derived()->encode_addU32(lhs_ref.part(0),
      ↪  rhs_ref.part(0), res_ref.part(0));
    case 64:
      return derived()->encode_addU64(lhs_ref.part(0),
      ↪  rhs_ref.part(0), res_ref.part(0));
    ...
 }
}
```

(b) Generating code for `arith.addi`

Figure 3: Adding backend support for the `arith.addi` MLIR operation

Snippets. To avoid implementing challenging aspects of a compiler backend, such as instruction selection, TPDE provides the *EncodeGen* tool. This tool takes a snippet file as an input, which contains tiny methods that specify the semantics of operations, and then uses Clang to convert these snippets into LLVM-IR. This LLVM-IR serves as input to the *EncodeGen* tool, which converts it into encoder functions that encode the instruction set architecture (ISA) instructions. Since the *EncodeGen* tool only requires input in LLVM-IR, the platform-agnostic snippets can be written in any programming language that can be compiled to LLVM-IR.

Figure 3 shows a simple example of how an arithmetic addition operation (`arith.addi`) can be specified through multiple snippets (one per supported bitwidth) and then later called in the source code. In Figure 3a, simple C methods are specified that add two unsigned integers for different bit widths. These snippets are then compiled into an encoder method, which can be called from the source code as shown in Figure 3b. Lastly, we just need to retrieve the input and result references, then call the newly generated encoder methods, depending on the bitwidths. Each time we encode an `arith.addi` instruction during compilation, we can now call

this `compile_arith_addi` method in our backend. A real implementation can often handle entire categories of operations (e.g., arithmetic operations) in one function through lookup tables containing encoder function pointers (instead of a switch statement).

Target-Specific Backends. Because TPDE encodes code snippets in a target-agnostic manner, the required target-specific infrastructure remains lightweight and maintainable. For a given architecture, the target-specific backend file primarily specifies calling convention details, low-level control-flow operations such as jumps and conditional branches, and the mechanics of resolving global symbols, including those reached through the global offset table (GOT), using platform-specific relocations. The GOT holds symbol addresses filled in by the dynamic linker at load time, so position-independent code can access a global through an indirect load rather than a hardcoded address.

A primary responsibility of this component is bridging the gap between high-level IRs and machine instructions. For instance, while the MLIR `arith` dialect provides a platform-independent abstraction for integer comparison predicates, the backend maps these semantics directly to the target ISA's conditional flags and jump instructions. To further optimize code generation, this layer also orchestrates target-specific instruction fusion, such as merging comparisons directly into subsequent conditional branches or sign extensions. While mapping these low-level semantics can be tedious, TPDE provides high-level builder helpers and abstractions for most of these operations, significantly simplifying the implementation of new architectural backends.

3.1 Supporting DB-specific Operations

One key benefit of custom compilation backends is that they can directly compile domain-specific semantics rather than having to construct them from general-purpose instructions. Traditionally, this required implementing these custom operations individually for all target platforms [7]. Our novel MLIR-TPDE approach avoids this by specifying domain-specific semantics in the snippets file and using TPDE tooling to convert them into target-agnostic encoder methods at build time. Thus, general-purpose and MLIR domain-specific dialects can be compiled to efficient machine code, thereby avoiding the complexity of target-specific instructions. For example, a database may want to support domain-specific operations such as hashing, approximate comparison for Umbra-style strings [24], and pointer tagging.

Figure 4 illustrates this approach using a database-specific operation that checks whether a 16-bit probabilistic filter stored in the upper bits of a pointer matches a provided hash value [3]. The C implementation directly encodes the operation's semantics. At compile time, TPDE automatically generates an encoder function that emits the corresponding machine code. As shown in Figure 4b, it suffices to prepare value references for the input arguments, the global tags table, and the result before invoking the generated encoder. While the majority of operations can be handled in this platform-agnostic manner, some operations, primarily (conditional) branches, require platform-specific implementations. However, even these remain relatively simple due to TPDE's helper functions.

```
// snippet for ptr_tag_matches
bool ptr_tag_matches(void* ptr, uint64_t hash, void* bloomMaskPtr) {
 uint64_t shiftAmount = 53;
 uint64_t slot = hash >> shiftAmount;
 uint64_t tag = ((uint16_t*) bloomMaskPtr)[slot];
 uint16_t entry = (uint16_t) (uintptr_t) ptr;
 uint16_t negatedEntry = entry ^ 0xFFFF;
 uint16_t anded = tag & negatedEntry;
 bool isMatch = anded == 0;
 return isMatch;
}
```

(a) C snippet for `ptr_tag_matches`

```
bool compile_ptr_tag_matches_op(PtrTagMatches op) {
 // prepare value references from op arguments
 auto hash_vr = this->val_ref(op.getHash());
 auto ptr_vr = this->val_ref(op.getPtr());
 // prepare value reference to global tags array
 ValuePartRef bloom_masks_array_vpr{..., 8, Config::GP_BANK};
 // prepare result reference
 auto res_ref = this->result_ref(op.getMatches());
 // call encoder generated from the snippet
 return derived()->encode_ptr_tag_matches(ptr_vr.part(0),
 ↪  hash_vr.part(0), std::move(bloom_masks_array_vpr),
 ↪  res_ref.part(0));
}
```

(b) Generating code for `ptr_tag_matches`

Figure 4: Adding compiler backend support for the custom `ptr_tag_matches` MLIR operation

4 REDUCING MLIR'S OVERHEAD

In this section, we discuss different techniques for reducing MLIR's own overhead, which adds latency even before invoking the compilation backend.

Context Initialization. One key idea of MLIR is to build a general-purpose framework for high-level compilers that can use both built-in and custom dialects. However, this also requires registering and initializing all required dialects, along with their operations, types, and attributes, when creating a new MLIRContext, i.e., every time a query is compiled. This registration and initialization phase adds around half a millisecond to the observable compilation latency. To reduce this overhead, we keep a pool of pre-initialized context objects that is refilled asynchronously whenever a new context is taken, thus moving context initialization out of the critical path.

Tuning MLIR. Since most uses of MLIR are not concerned with low-latency compilation, some configurations are set by default to perform additional optimizations or verifications: MLIR's general mechanism for applying rewrite patterns defaults to several expensive simplification steps that did not lead to significant changes. Similarly, MLIR's built-in canonicalization pass can be tuned not to perform expensive region simplification by supplying a corresponding configuration. Additionally, MLIR performs expensive verification steps after each IR transformation. While this is valuable for development and testing (i.e., similar to asserts), disabling them for production significantly reduces compilation times.

Proactive Folding. Instead of relying on general canonicalization passes provided by the MLIR ecosystem to clean up the IR after transformations, we can directly apply the folding proactively, rewriting an operation into a simpler equivalent at the moment it is constructed. By using MLIR's `createOrFold` when constructing

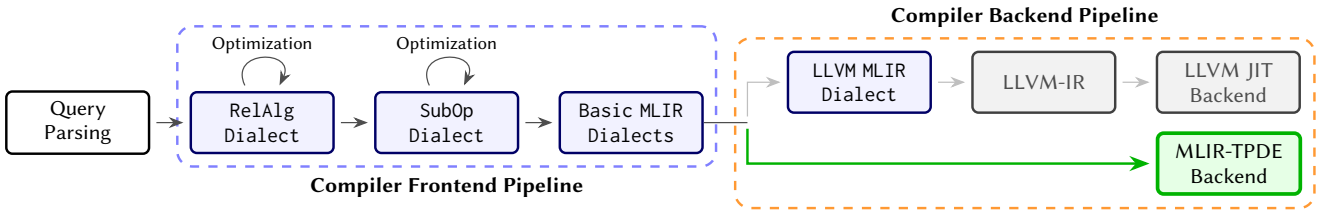


Figure 5: Overall query compilation process in LingoDB. The Compiler Frontend Pipeline leverages MLIR for progressive lowering with internal optimizations, branching to either a legacy LLVM path or the novel MLIR-TPDE backend.

new operations, redundant pairs of boxing and unboxing operations on nullable values or tuples are eliminated immediately at creation time. Boxing wraps a raw value in a representation that carries extra information, such as a null flag for a nullable value or the element layout of a tuple, and unboxing recovers the original value, so a box immediately followed by an unbox cancels out. This approach preserves the beneficial effects of canonicalization for the relevant operations while avoiding the large overhead of a full canonicalization pass that attempts to simplify a larger set of operations. When additional cleanup is still required, lightweight, specialized passes that apply only a small, targeted set of patterns can be used instead of general canonicalization.

5 IMPLEMENTATION IN LINGODB

We now discuss the integration of both MLIR-TPDE and the described techniques for speeding up MLIR into LingoDB [16], our MLIR-based, open source query engine. While LingoDB shares many concepts with foundational compiling databases like HyPer [17] or Umbra [24], LingoDB implements query optimization using MLIR compiler passes over high-level relational dialects and query compilation through a layered, multistep process. At the end of this lowering process, the final MLIR module is compiled to machine code via LLVM. This difference significantly reduces engineering effort, as previous work has shown [16].

LingoDB's Optimization and Compilation Pipeline. Figure 5 shows the overall code-generating pipeline of LingoDB. After parsing the query statement, LingoDB transforms the input into the high-level `RelAlg` dialect, which represents common database operators such as joins and aggregations. After applying optimization passes such as predicate pushdown and join order optimizations, relational operators are transformed into simpler sub-operators. This `SubOp` dialect consists of foundational building blocks with explicit state and control flow, while its optimization passes try to parallelize the given query or to avoid duplicate pipelines [14]. With each lowering step, the compiler optimizes the code, and the operations in this IR transform into simpler ones. At the end of the compiler frontend, the MLIR module consists of basic blocks in MLIR-builtin dialects (`func, scf, arith`), such as function calls, control flow, and basic arithmetic operations, as well as a custom low-level `util` dialect. This code can then be passed to the existing LLVM-based MLIR JIT compiler pipeline. This pipeline first lowers the module to MLIR's `llvm` dialect, which can then be translated into LLVM-IR. Afterward, LLVM is used to compile LLVM-IR into machine code.

Integration of MLIR-TPDE. We integrated MLIR-TPDE as an alternative LingoDB backend that can compile the LingoDB-specific mix of low-level dialects (`func, scf, arith, util`). Before invoking MLIR-TPDE, we first run two MLIR passes that convert structured control-flow (e.g., if statements, for loops) into basic blocks (`cf` dialect), and further rewrite tuple-related operations to avoid implementing support for non-scalar types in MLIR-TPDE. As discussed in Section 3.1, we add additional snippets for the custom operations of the `util` dialect.

Reducing MLIR overhead. Additionally, we implemented the techniques described in Section 4 to reduce the MLIR-intrinsic overhead. More specifically, we keep pre-initialized context objects, use a modified configuration for pattern application and canonicalization passes, and eliminate cleanup passes (canonicalization and common subexpression elimination) passes in favor of direct folding.

Orthogonally, we also reduce the amount of MLIR code generated in early steps of the pipeline, which decreases the workload for all following lowering passes while preserving execution quality, for example, by using vectorized filters on data chunks instead of generating code for it [4, 18], or reducing unnecessary null checks and unboxing operations. Additionally, using MLIR allows for easily introducing new low-level, fused operations for common use cases, which can significantly reduce the number of operations that need to be processed by subsequent transformation passes. For example, we introduced fused operations for (1) checking if the tag in a tagged pointer matches a hash value, (2) setting or checking specific bits in integers, and (3) loading and storing elements from struct pointers, without explicitly calculating the element's address.

6 EVALUATION

In this section, we evaluate the implementation of the proposed improvements and the TPDE-based compilation backend in LingoDB. First, we evaluate the new MLIR-TPDE backend by comparing it against existing LLVM-based LingoDB backends and Hyper's compilation backend. Second, we evaluate the effect of different optimization techniques on the latency of high-level code generation and compare it with the latency of comparable steps in Hyper. Finally, we compare LingoDB's end-to-end query latencies against state-of-the-art systems, showing that LingoDB is now end-to-end competitive, even for smaller datasets of around 1 GB. For the experiments in this section, each query was run 5 times for warmup, and the median execution and compilation times are reported across 20 measured runs. All experiments were run multithreaded on

Table 1: Average compilation latencies and execution times in ms of our three backend variants

x86	Backend	MLIR	Opt.	Code Gen.	∑	Execution (SF=1/SF=10)
TPC-H	MLIR-TPDE	0.28	0.00	0.29	0.57	7.44 / 77.20
	LLVM O0	4.43	0.00	6.45	10.88	8.21 / 76.79
	LLVM Opt	4.51	2.38	19.28	26.17	7.56 / 66.10
TPC-DS	MLIR-TPDE	0.84	0.00	0.68	1.53	11.16 / 94.97
	LLVM O0	10.85	0.00	14.83	25.68	12.26 / 95.76
	LLVM Opt	10.72	8.31	50.54	69.57	11.20 / 81.35
aarch	Backend	MLIR	Opt.	Code Gen.	∑	Execution (SF=1/SF=10)
TPC-H	MLIR-TPDE	0.31	0.00	0.31	0.62	6.94 / 79.50
	LLVM O0	4.98	0.00	6.70	11.69	7.96 / 88.53
	LLVM Opt	5.12	2.38	20.31	27.81	6.31 / 62.84
TPC-DS	MLIR-TPDE	0.91	0.00	0.73	1.64	10.16 / 99.50
	LLVM O0	12.13	0.00	14.58	26.72	11.56 / 106.96
	LLVM Opt	12.13	8.36	51.33	71.83	9.69 / 81.21

both x86-64 and aarch64, with x86-64 used unless explicitly annotated otherwise. All x86-64 measurements were performed on a m7a.x4large instance running Ubuntu 24.04 on an AMD EPYC 9R14 CPU with 16 virtual cores and 64 GiB of main memory. The aarch64 experiments were executed on a Graviton 4 instance (m8g.4xlarge), with 16 cores at 2.7 GHz, 64 GiB of main memory, and Ubuntu 24.04. The following software versions were used: Umbra 26.02, Tableau Hyper 0.0.22106, and DuckDB 1.4.3.

6.1 MLIR-TPDE

In this experiment, we evaluate the compilation latency and execution speed of our novel MLIR-TPDE backend.

In-system comparison. We compare our novel MLIR-TPDE backend with our two existing LLVM compiler backends: LLVM O0 and LLVM Opt. The LLVM O0 backend configuration aims for fast compilation, whereas LLVM Opt applies a selection of optimization passes that allow it to generate code comparable to the O2 configuration of LLVM.

Table 1 lists the average times for the required MLIR preparation passes, optimization passes, machine code generation, and the time for executing the generated code. Overall, we observe that MLIR-TPDE can generate code 16x to 19x faster than LLVM O0, while achieving execution speeds that typically fall between LLVM O0 and LLVM Opt. In the MLIR transformation phase, MLIR-TPDE reduces the total time required by between 13x and 16x compared to both LLVM backends. Similarly, MLIR-TPDE requires 20x and 66x less time than LLVM O0 and LLVM Opt to generate the machine code. Interestingly, for x86 and scale factor 1, the MLIR-TPDE backend outperforms even LLVM Opt, while in all other cases execution times either roughly match those of LLVM O0 or are between LLVM O0 and LLVM Opt.

Most of the benefits of using MLIR-TPDE can be attributed to the following attributes: (1) MLIR-TPDE avoids the costly conversion to an IR of the existing compiling backend by directly using the IR at hand, (2) it is able to generate machine code 20x quicker than LLVM O0, and lastly, it (3) generates machine code, that executes similar or up to 13% faster than LLVM O0, but on large datasets up to 21% slower than LLVM Opt.

Table 2: Backend Average Latency Comparison (ms)

	LingoDB			Hyper
Benchmark	**Lowerings**	**Code Gen.**	**∑**	**∑**
TPC-H	0.29	0.29	0.58	1.27
TPC-DS	0.86	0.69	1.55	2.61

Table 3: Effect of different techniques on the high-level compilation latency: average latencies for the different phases in ms

	Optimization	SQL	Opt.	Lowerings	∑
TPC-H	-	1.0	1.4	7.0	9.4
	NoVerification	1.0 *+2%*	1.1 *-20%*	5.2 *-26%*	7.3 *-22%*
	AsyncContext	0.6 *-45%*	1.2 *+5%*	5.2 *+2%*	7.0 *-4%*
	NoCleanup	0.5 *-6%*	1.2 *+0%*	3.7 *-30%*	5.4 *-23%*
	Patterns	0.5 *+0%*	1.0 *-17%*	2.8 *-23%*	4.3 *-20%*
TPC-DS	-	1.6	3.3	17.8	22.7
	NoVerification	1.6 *+0%*	2.6 *-22%*	12.9 *-27%*	17.1 *-25%*
	AsyncContext	1.2 *-28%*	2.6 *+2%*	13.0 *+0%*	16.8 *-2%*
	NoCleanup	1.2 *+0%*	2.6 *+0%*	8.6 *-34%*	12.4 *-26%*
	Patterns	1.1 *-1%*	2.1 *-19%*	6.2 *-28%*	9.5 *-23%*

Comparison against Hyper. We compare the novel MLIR-TPDE compilation latencies to those of the Hyper database.

In Table 2, the average compilation latency of LingoDB's MLIR-TPDE backend is compared to that of Hyper. Interestingly, LingoDB is able to outperform Hyper by 41% to 54% depending on the benchmark. To summarize, by using MLIR-TPDE, LingoDB can generate high-quality code faster than custom backends like Hyper, even though Hyper reportedly uses a custom, optimized IR representation.

6.2 MLIR Overhead

In Section 4, we explain the core techniques we used to speed up MLIR towards low-latency compilation. In this ablation study, we quantify the effects of the different configurations.

In Table 3, we measured the average compilation latencies across the high-level compilation phases in TPC-H and TPC-DS. The *SQL* phase measures the time to generate the basic MLIR module starting from the input SQL query. Then *Opt.* denotes the time spent inside our custom MLIR dialects optimizing the MLIR, and lastly, *Lowerings* denotes the time spent lowering between dialects. Overall, *NoVerification, NoCleanup* and *Patterns* are the most effective optimizations, each reducing the total time by around 20% to 26%. Disabling MLIR's verifications after transformations (*NoVerification*) helps effectively reduce the *Opt.* and *Lowerings* phases by around 24%. Similarly, deactivating simplification steps by tuning MLIR's pattern application helps reduce the *Opt.* and *Lowerings* phases by around 21%. Further reducing the number of passes by avoiding additional cleanup passes (*NoCleanup*) leads to a similar reduction in the *Lowerings* phase and total time. While pre-creating the MLIR contexts for incoming queries (*AsyncContext*) helps cut the SQL phase by 28% to 45%, it only reduces the total time by

Table 4: Frontend Engine Average Latency Breakdown (ms)

	LingoDB				Hyper		
Benchmark	**SQL**	**Opt.**	**Lowerings**	**∑**	**SQL & Opt.**	**IR Gen.**	**∑**
TPC-H	0.52	0.97	2.83	4.32	0.97	0.87	1.84
TPC-DS	1.14	2.12	6.21	9.47	2.06	1.66	3.72

around 3% as the lowering phase dominates the total time required. To summarize, by applying these four optimizations above, we can effectively halve the total high-level compilation phase.

Comparison against Hyper. To further put our results into perspective, we compare LingoDB's compiler frontend times to those of Hyper. We define the frontend compilation latency as the time between receiving the SQL query until the query is optimized, and an imperative representation is generated that can be passed to a compiler backend.

In Table 4, we compare the average compiler frontend times between LingoDB and Hyper. Overall, Hyper completes this frontend phase 2.3x to 2.5x faster than LingoDB. While the *SQL* and *Opt.* combined are around 1.5x faster in Hyper, most of the difference comes from the lowerings implemented in MLIR. Compared to Hyper, LingoDB takes around 3.5x longer to lower the optimized code to a form usable by a compiler backend.

In contrast to the faster backend compiler latency (see Section 6.1), LingoDB suffers here from its design decisions and the generality of MLIR. As explained in Section 5, LingoDB first transforms the parsed SQL statement into a high-level *RelAlg* dialect that represents database operations such as joins and table scans. This *RelAlg* MLIR module is then optimized and eventually lowered to a mid-level MLIR dialect called *SubOp* [14]. Yet again, optimizations are applied, and then the module is finally lowered to low-level MLIR dialects that can be passed to a compiler backend. While this abstraction approach simplifies optimization, as the developer can reason about optimizations in high- and mid-level dialects, and reduces the engineering effort [16], it comes at the cost of higher frontend times. The extensibility of the generic compiler framework MLIR and these developer-friendly abstractions of LingoDB are creating a significant frontend overhead.

6.3 End-to-End Query Latency

In this experiment, we compare LingoDB using the existing LLVM Opt and the novel MLIR-TPDE backend end-to-end against two compiling database systems, Hyper and Umbra, and DuckDB as a vectorized system. Both LingoDB backends use the techniques to speed up MLIR (Section 4), isolating the impact of our novel MLIR-TPDE backend. The LLVM backend uses a configuration that emits code comparable to -O2.

Figure 6 shows the sum of all query latencies of TPC-H and TPC-DS (SF1 and SF10) queries on x86 and aarch64. Overall, we observe that with our proposed changes, LingoDB has competitive performance across all benchmarks and scaling factors. When we compare the two LingoDB backends, MLIR-TPDE compiles 7x faster. Regarding execution speed, MLIR-TPDE and LLVM emit similar, high-performance code when using SF1, whereas in the SF10 benchmark, we observe up to 26% slower execution times with the MLIR-TPDE backend. However, when comparing total query latencies, MLIR-TPDE completes the SF1 benchmarks around 3.2x faster, while the SF10 benchmarks finish at least 20% faster. Compared to Hyper, LingoDB has faster execution times on all four benchmark datasets. While LingoDB still suffers from up to 65% higher compilation times than Hyper, it can outperform Hyper across all benchmarks due to faster execution, especially in the x86 TPC-DS SF10 benchmark. Compared to the vectorized database DuckDB, LingoDB achieves lower total query latency, effectively amortizing the compilation overhead by executing more efficient code. Nevertheless, Umbra can outperform all tested databases, achieving the smallest compilation and execution times.

Table 5: Implementation Effort Breakdown. The majority of the logic is target-independent, with minimal code required for specific architectures (x86/aarch64).

Category	**LOC**
Target-independent adapter	1922
LingoDB Backend	456
Snippets	242
x86-specific code	249
aarch64-specific code	230
Total	**3099**

6.4 Implementation Effort

Custom query compiler backends are frequently notoriously complex and difficult to maintain. To avoid this complexity, other query engines use existing compiling backends at the cost of higher compilation latency, e.g., due to costly IR conversions. TPDE allows us to get the best of both worlds: (1) not having to reimplement complicated parts (e.g., register allocation, manually generating code for custom operations) or (2) having to convert to a compiler backend-specific IR while (3) being able to directly compile custom operations.

Table 5 shows the lines of code (LOC) required for implementing the different parts of MLIR-TPDE in LingoDB. Overall, we can see that it only took less than 3.1k lines of code to fully implement the entire backend. The largest component is the platform-independent adapter (1922 lines), which bridges MLIR and TPDE, and compiles most operations through encoders generated from the Snippets (242 lines). The amount of target-specific code is relatively small (249 for x86), and a larger part of it stems from a fused implementation of comparisons and jumps, for which assembly instructions are emitted manually via TPDE. Finally, a few more lines are required for an integration into LingoDB that involves lowering the `scf` dialect, error handling, and other smaller details. As already hinted, adding support for a new operation is usually as simple as writing a small snippet in C and calling the generated encoder with a bit of boilerplate.

While LOC does not directly determine whether a system is maintainable, we still want to compare it with Umbra's compiler backend. Previous work quantified the LOC of the x86 and aarch64 backends in Umbra at 5.3k and 8k, respectively [7]. In total, that indicates that the complete backend consists of 13.3k LOC. Compared

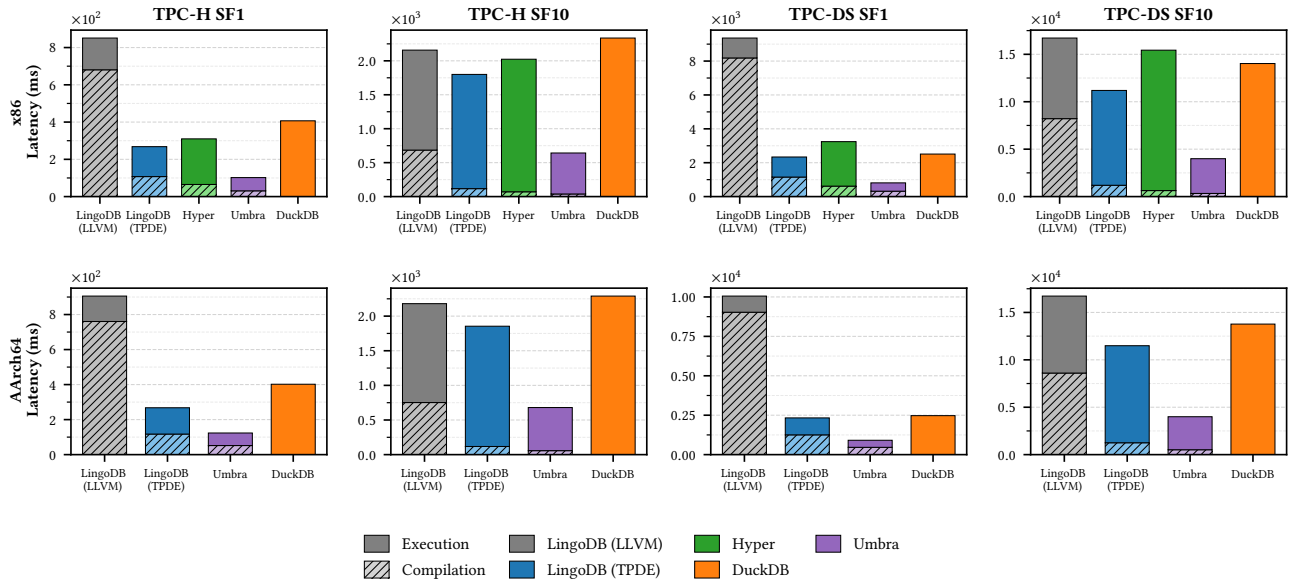


Figure 6: End-to-end Latency on x86 and AArch64. LingoDB (MLIR-TPDE) significantly reduces compilation overhead compared to the LLVM backend.

to our novel approach, we can achieve similar backend compilation times, while requiring less than a quarter of Umbra's backend LOC. This is achieved by delegating most of the complexity to TPDE, which is actively maintained with a core code base of around 17k lines of code.

7 RELATED WORK

In this paper, we present our novel MLIR-TPDE query compiler backend in LingoDB and explain the core techniques to make MLIR usable in latency-critical systems. In the following, we will discuss related work and contrast it with our contribution.

Low-Latency Query Compilation. Several systems have explored low-latency query compilation, like HyPer [17], Umbra [24], Mutable [10] and NebulaStream/Nautilus [8, 27]. To achieve sub-millisecond compilation latencies, they have to bypass the heavy, general-purpose compiler frameworks like LLVM. Custom IRs, such as Umbra's Flying Start [19], achieve extreme compilation speed by mapping closely to machine code. However, this requires writing and maintaining a custom compiler backend from scratch, which is notoriously difficult to build and to port to new hardware architectures [7]. Our approach of combining the flexibility and maintainability of MLIR with the low-latency compiler backend framework TPDE achieves similar compilation latencies while significantly reducing the engineering effort.

Hiding Compilation Latency. Others have proposed approaches that avoid the engineering effort of the just-discussed low-latency compilation methods. For example, Inkfuse [26] decomposes the query into sub-operators that can be either compiled with LLVM or executed directly with vectorized functions generated on the fly. Amazon Redshift [1] relies heavily on a distributed compilation service that caches frequently compiled code for commonly used query fragments.

Adaptive Query Compilation. Orthogonally to our approach, previous work has explored switching from interpreted or fast-emitted code to optimized code during execution to improve the performance of long-running queries [9, 12, 19, 20]. Adaptive query compilation was pioneered by HyPer [20], while this approach is able to emit bytecode quickly, it suffered from slow execution speed due to interpretation overhead. Umbra's Flying Start is able to bridge this gap by emitting compiled code directly to native instructions [19]. However, as shown in Section 6.4, this approach requires more than 4x more LOC (x86 and aarch64) than our solution [7].

Recent work attempt to solve this maintainability gap by using off-the-shelf tooling like virtual machines or WebAssembly runtimes. For example, Haffner et al. use Google's WebAssembly runtime *V8*, which allows for fast compilation using *Liftoff* and allows replacing it on the fly with optimized code produced by *TurboFan*, in their database *mutable* [11, 12]. However, it is unable

to process larger datasets due to only having 4 GiB of addressable linear memory inside the WebAssembly module.

Reducing MLIR's Compilation Times. Over the past few years, MLIR [22] has gained traction in the database community as a way to construct modular, easily maintainable query compilers [8, 16]. With Nautilus [8], Grulich et al. proposed a novel tracing-based approach for generating a query-specific IR from C++ operator implementations in a single pass. Experiments from previous work in LingoDB showed that bypassing the LLVM bottleneck can drastically reduce query latency. But this template-based backend for MLIR suffered from 1x to 3x slower execution times compared to the LLVM backend [6]. In contrast, our novel MLIR-TPDE backend generates high-quality code, resulting in execution speeds similar to those of the LLVM backend.

8 DISCUSSION: REUSABILITY

We now discuss the reusability of our novel MLIR-TPDE compiler backend (Section 3) and the techniques proposed to speed up MLIR (Section 4), both of which have already been released as open source via the LingoDB project.

Techniques such as pre-initializing MLIR context objects or disabling verification after IR transformations move work out of the critical path. Additionally, proactive folding and deliberately reducing the amount of generated code further reduce compilation latency. Together, these techniques make MLIR viable for low-latency compilation scenarios and can be adopted by virtually any MLIR-based system. The MLIR-TPDE backend is likewise reusable. As explained in Section 3, LingoDB first lowers from system-specific high-level MLIR dialects to the lower-level builtin dialects (`arith`, `cf`, `func`). Any system that implements lowering for these core dialects can directly use our TPDE adapter for this subset of MLIR and increase coverage by adding snippets with modest effort. The vast majority of the target-specific code concerns mapping MLIR control-flow operations (conditional branches and jumps) to the target architecture and is therefore generally useful. For database systems and similar workloads, additional reusable components such as support for Umbra-style 128-bit variable-length strings [24], bit manipulation, and hash table-specific operations further reduce the engineering effort required to adapt the backend. In combination, the MLIR-level tunings and the reusable MLIR-TPDE backend substantially reduce both compilation latency and implementation effort for low-latency, MLIR-based code generation.

9 CONCLUSION

This paper addressed the compilation-latency bottleneck of MLIR-based query engines with two contributions. First, the new MLIR-TPDE compilation backend bypasses the existing LLVM lowering and compilation pipeline and is significantly faster in generating efficient code. Second, we discuss several techniques that can halve MLIR's intrinsic overhead. Integrated into LingoDB, this lets LingoDB reach end-to-end latencies competitive with state-of-the-art systems like Hyper and DuckDB, while keeping the implementation effort low.

ACKNOWLEDGMENTS

This work was supported by the German Federal Ministry of Education and Research (BMBF) under the Software Campus program (project REALDRAGON, grant number 01IS23069), and through a Google PhD Fellowship.

REFERENCES

[1] Nikos Armenatzoglou, Sanuj Basu, Naga Bhanoori, Mengchu Cai, Naresh Chainani, Kiran Chinta, Venkatraman Govindaraju, Todd J. Green, Monish Gupta, Sebastian Hillig, Eric Hotinger, Yan Leshinksy, Jintian Liang, Michael McCreedy, Fabian Nagel, Ippokratis Pandis, Panos Parchas, Rahul Pathak, Orestis Polychroniou, Foyzur Rahman, Gaurav Saxena, Gokul Soundararajan, Sriram Subramanian, and Doug Terry. 2022. Amazon Redshift Re-invented. In *SIGMOD '22: International Conference on Management of Data, Philadelphia, PA, USA, June 12 - 17, 2022*, Zachary G. Ives, Angela Bonifati, and Amr El Abbadi (Eds.). ACM, 2205–2217. https://doi.org/10.1145/3514221.3526045

[2] Alexander Behm and Shoumik Palkar. 2022. Photon: A High-Performance Query Engine for the Lakehouse. In *CIDR*. www.cidrdb.org.

[3] Altan Birler, Tobias Schmidt, Philipp Fent, and Thomas Neumann. 2024. Simple, Efficient, and Robust Hash Tables for Join Processing. In *Proceedings of the 20th International Workshop on Data Management on New Hardware, DaMoN 2024, Santiago, Chile, 10 June 2024*, Carsten Binnig and Nesime Tatbul (Eds.). ACM, 4:1–4:9. https://doi.org/10.1145/3662010.3663442

[4] Peter Boncz, Marcin Zukowski, and Niels Nes. 2005. MonetDB/X100: Hyper-Pipelining Query Execution. In *Second Biennial Conference on Innovative Data Systems Research, CIDR 2005, Asilomar, CA, USA, January 4-7, 2005, Online Proceedings*. www.cidrdb.org, 225–237. http://cidrdb.org/cidr2005/papers/P19.pdf

[5] Patrick Damme, Marius Birkenbach, Constantinos Bitsakos, Matthias Boehm, Philippe Bonnet, Florina M. Ciorba, Mark Dokter, Pawel Dowgiallo, Ahmed Eleliemy, Christian Faerber, Georgios I. Goumas, Dirk Habich, Niclas Hedam, Marlies Hofer, Wenjun Huang, Kevin Innerebner, Vasileios Karakostas, Roman Kern, Tomaz Kosar, Alexander Krause, Daniel Krems, Andreas Laber, Wolfgang Lehner, Eric Mier, Marcus Paradies, Bernhard Peischl, Gabrielle Poerwawinata, Stratos Psomadakis, Tilmann Rabl, Piotr Ratuszniak, Pedro Silva, Nikolai Skuppin, Andreas Starzacher, Benjamin Steinwender, Ilin Tolovski, Pinar Tözün, Wojciech Ulatowski, Yuanyuan Wang, Izajasz P. Wrosz, Ales Zamuda, Ce Zhang, and Xiaoxiang Zhu. 2022. DAPHNE: An Open and Extensible System Infrastructure for Integrated Data Analysis Pipelines. In *CIDR*. www.cidrdb.org.

[6] Florian Drescher and Alexis Engelke. 2024. Fast Template-Based Code Generation for MLIR. In *Proceedings of the 33rd ACM SIGPLAN International Conference on Compiler Construction, CC 2024, Edinburgh, United Kingdom, March 2-3, 2024*, Gabriel Rodríguez, P. Sadayappan, and Aravind Sukumaran-Rajam (Eds.). ACM, 1–12. https://doi.org/10.1145/3640537.3641567

[7] Ferdinand Gruber, Maximilian Bandle, Alexis Engelke, Thomas Neumann, and Jana Giceva. 2023. Bringing Compiling Databases to RISC Architectures. *Proc. VLDB Endow.* 16, 6 (2023), 1222–1234. https://doi.org/10.14778/3583140.3583142

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

[9] Tim Gubner and Peter Boncz. 2022. Excalibur: A Virtual Machine for Adaptive Fine-grained JIT-Compiled Query Execution based on VOILA. *Proc. VLDB Endow.* 16, 4 (2022), 829–841. https://doi.org/10.14778/3574245.3574266

[10] Immanuel Haffner and Jens Dittrich. 2023. mutable: A Modern DBMS for Research and Fast Prototyping. In *13th Conference on Innovative Data Systems Research, CIDR 2023, Amsterdam, The Netherlands, January 8-11, 2023*. www.cidrdb.org. https://vldb.org/cidrdb/2023/mutable-a-modern-dbms-for-research-and-fast-prototyping.html

[11] Immanuel Haffner and Jens Dittrich. 2023. mutable: A Modern DBMS for Research and Fast Prototyping. In *13th Conference on Innovative Data Systems Research, CIDR 2023, Amsterdam, The Netherlands, January 8-11, 2023*. www.cidrdb.org. https://vldb.org/cidrdb/2023/mutable-a-modern-dbms-for-research-and-fast-prototyping.html

[12] Immanuel Haffner and Jens Dittrich. 2023. A simplified Architecture for Fast, Adaptive Compilation and Execution of SQL Queries. In *Proceedings 26th International Conference on Extending Database Technology, EDBT 2023, Ioannina, Greece, March 28-31, 2023*, Julia Stoyanovich, Jens Teubner, Nikos Mamoulis, Evaggelia Pitoura, Jan Mühlig, Katja Hose, Sourav S. Bhowmick, and Matteo Lissandrini (Eds.). OpenProceedings.org, 1–13. https://doi.org/10.48786/EDBT.2023.01

[13] Michael Jungmair. 2025. LingoDB-CT: Understanding LingoDB's Inner Workings. In *SIGMOD Conference Companion*. ACM, 139–142.

[14] Michael Jungmair and Jana Giceva. 2023. Declarative Sub-Operators for Universal Data Processing. *Proc. VLDB Endow.* 16, 11 (2023), 3461–3474. https://doi.org/10.14778/3611479.3611539

[15] Michael Jungmair and Jana Giceva. 2025. Towards Designing Future-Proof Data Processing Systems. *Proc. VLDB Endow.* 18, 11 (2025), 3988–3995.

[16] Michael Jungmair, André Kohn, and Jana Giceva. 2022. Designing an Open Framework for Query Optimization and Compilation. *Proc. VLDB Endow.* 15, 11 (2022), 2389–2401. https://doi.org/10.14778/3551793.3551801

[17] Alfons Kemper and Thomas Neumann. 2011. HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots. In *Proceedings of the 27th International Conference on Data Engineering, ICDE 2011, April 11-16, 2011, Hannover, Germany*, Serge Abiteboul, Klemens Böhm, Christoph Koch, and Kian-Lee Tan (Eds.). IEEE Computer Society, 195–206. https://doi.org/10.1109/ICDE.2011.5767867

[18] Timo Kersten, Viktor Leis, Alfons Kemper, Thomas Neumann, Andrew Pavlo, and Peter Boncz. 2018. Everything You Always Wanted to Know About Compiled and Vectorized Queries But Were Afraid to Ask. *Proc. VLDB Endow.* 11, 13 (2018), 2209–2222. https://doi.org/10.14778/3275366.3275370

[19] 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. https://doi.org/10.1007/S00778-020-00643-4

[20] André Kohn, Viktor Leis, and Thomas Neumann. 2018. Adaptive Execution of Compiled Queries. In *34th IEEE International Conference on Data Engineering, ICDE 2018, Paris, France, April 16-19, 2018*. IEEE Computer Society, 197–208. https://doi.org/10.1109/ICDE.2018.00027

[21] Konstantinos Krikellas, Stratis Viglas, and Marcelo Cintra. 2010. Generating code for holistic query evaluation. In *Proceedings of the 26th International Conference on Data Engineering, ICDE 2010, March 1-6, 2010, Long Beach, California, USA*, Feifei Li, Mirella M. Moro, Shahram Ghandeharizadeh, Jayant R. Haritsa, Gerhard Weikum, Michael J. Carey, Fabio Casati, Edward Y. Chang, Ioana Manolescu, Sharad Mehrotra, Umeshwar Dayal, and Vassilis J. Tsotras (Eds.). IEEE Computer Society, 613–624. https://doi.org/10.1109/ICDE.2010.5447892

[22] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. 2021. MLIR: Scaling Compiler Infrastructure for Domain Specific Computation. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 2–14. https://doi.org/10.1109/CGO51591.2021.9370308

[23] Vladimir N. Makarov. 2024. MIR: A lightweight JIT compiler based on MIR (Medium Internal Representation) and C11 JIT compiler and interpreter based on MIR. https://github.com/vnmakarov/mir. Version 1.0.0, accessed 2026-05-22.

[24] Thomas Neumann and Michael J. Freitag. 2020. Umbra: A Disk-Based System with In-Memory Performance. In *10th Conference on Innovative Data Systems Research, CIDR 2020, Amsterdam, The Netherlands, January 12-15, 2020, Online Proceedings*. www.cidrdb.org. https://vldb.org/cidrdb/2020/umbra-a-disk-based-system-with-in-memory-performance.html

[25] Tobias Schwarz, Tobias Kamm, and Alexis Engelke. 2026. TPDE: A Fast Adaptable Compiler Back-End Framework. In *IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2026, Sydney, Australia, January 31 - Feb. 4, 2026*, Stephen M. Blackburn, Albert Cohen, and Timothy M. Jones (Eds.). IEEE, 427–439. https://doi.org/10.1109/CGO68049.2026.11395208

[26] Benjamin Wagner, André Kohn, Peter Boncz, and Viktor Leis. 2024. Incremental Fusion: Unifying Compiled and Vectorized Query Execution. In *40th IEEE International Conference on Data Engineering, ICDE 2024, Utrecht, The Netherlands, May 13-16, 2024*. IEEE, 462–474. https://doi.org/10.1109/ICDE60146.2024.00042

[27] Steffen Zeuch, Ankit Chaudhary, Bonaventura Del Monte, Haralampos Gavriilidis, Dimitrios Giouroukis, Philipp M. Grulich, Sebastian Breß, Jonas Traub, and Volker Markl. 2020. The NebulaStream Platform for Data and Application Management in the Internet of Things. In *10th Conference on Innovative Data Systems Research, CIDR 2020, Amsterdam, The Netherlands, January 12-15, 2020, Online Proceedings*. www.cidrdb.org. https://vldb.org/cidrdb/2020/the-nebulastream-platform-data-and-application-management-for-the-internet-of-things.html