Designing an Open Framework for Query Optimization and Compilation

Michael Jungmair
Technische Universität München
jungmair@in.tum.de

André Kohn
Technische Universität München
kohna@in.tum.de

Jana Giceva
Technische Universität München
jana.giceva@in.tum.de

ABSTRACT

Since its invention, data-centric code generation has been adopted for query compilation by various database systems in academia and industry. These database systems are fast but maximize performance at the expense of developer friendliness, flexibility, and extensibility. Recent advances in the field of compiler construction identified similar issues for domain-specific compilers and introduced a solution with MLIR, a generic infrastructure for domain-specific dialects.

We propose a layered query compilation stack based on MLIR with open intermediate representations that can be combined at each layer. We further propose moving query optimization into the query compiler to benefit from the existing optimization infrastructure and make cross-domain optimization viable. With LingoDB, we demonstrate that the used approach significantly decreases the implementation effort and is highly flexible and extensible. At the same time, LingoDB achieves high performance and low compilation latencies.

PVLDB Reference Format:
Michael Jungmair, André Kohn, and Jana Giceva. Designing an Open Framework for Query Optimization and Compilation. PVLDB, 15(11): 2389 - 2401, 2022.
doi:10.14778/3551793.3551801

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://doi.org/10.5281/zenodo.6786922.

1 INTRODUCTION

Despite the wide adoption of data-centric code generation [22] in academia [5, 10, 23] and industry [7, 33, 35], modern query compilers face a number of issues. First, developing a query compiling system is still a non-trivial undertaking. This is partly due to a missing consensus on how query compilers should be built [11, 13, 32, 34], but also due to a lack of an open framework for building query compilers. As a result, new systems often start from scratch and reinvent solutions for many reoccurring tasks like the interaction with low-level compiler frameworks (e.g., LLVM [16]), type systems, and runtimes. Second, compiling query engines are usually not designed with flexibility as the main feature and instead aim for maximum performance. This makes it difficult to integrate novel ideas into an existing query compiler. Especially incremental research in this field that targets heterogeneous hardware platforms [27] or advanced aggregation operators [15] are often built with very different system architectures and cannot be easily combined. Evaluating such ideas often requires building dedicated research prototypes, which increases the barriers to adoption and quickly leads to incomparable system performances. Finally, current compiling database systems are also hard to extend beyond relational workloads. Other data processing domains cannot be easily integrated, and cross-domain optimization on a higher abstraction level is often left out despite its potential [9].

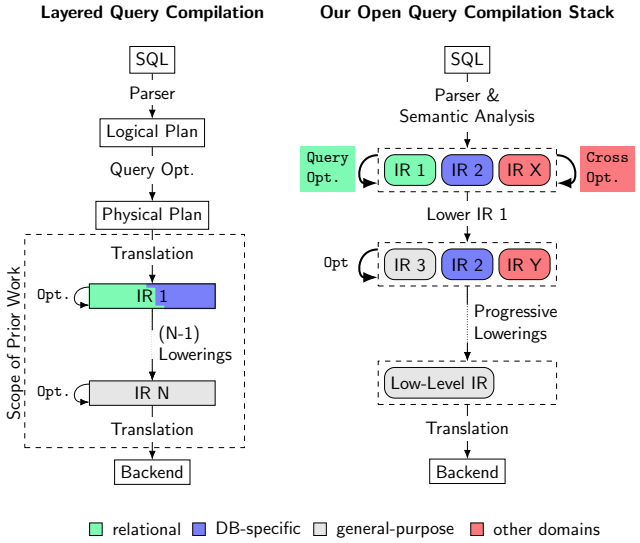

Figure 1: Our proposal of an *open* query compilation stack. It enhances prior work on layered query compilation with two major ideas: 1) Introducing open IRs, designed to be combined with other IRs, and 2) implementing query optimization as compiler passes.

These issues are not unique to database architectures and also affect other domain-specific compilers. Machine-learning frameworks, for example, are composed of various representations that target a wide range of runtime systems [1]. Their solution to this problem is called Multi-Level Intermediate Representation (MLIR), which proposes a novel compiler infrastructure [17]. MLIR offers a unified intermediate representation that can express domain-specific dialects (a set of custom operations that operate on custom types) on different abstraction levels. This allows for representing low-level, assembly-like languages like the LLVM IR and high-level dataflow graphs (e.g., Tensorflow graphs) within the same intermediate representation. The main achievement of MLIR is, therefore,

Proceedings of the VLDB Endowment, Vol. 15, No. 11 ISSN 2150-8097.
doi:10.14778/3551793.3551801

the reusability of dialects, besides providing a common language infrastructure. It encourages to reuse existing dialects as building blocks which simplifies the development, evaluation, and integration of new ideas.

In this paper, we address the issues raised above by proposing an open query compilation stack in the spirit of MLIR. To make query compilers more flexible, Shaikhha et al. [32] proposed using multiple layers of intermediate representations (IR), as depicted on the left side of Figure 1. However, layered query compilation often leads to high implementation efforts as multiple IRs and lowerings must be implemented. Furthermore, existing implementations suffer from high compilation latencies. In this work, we make layered query compilation more practical by reducing the implementation effort and the compilation latency. To both increase the flexibility even further and to enable true extensibility, we propose using multiple, *open*, combinable IRs on each layer, as shown on the right side of Figure 1. This increases the reusability of database-specific dialects and allows for directly integrating existing IRs of other domains. As shown in Figure 1, we can extend our query compiler with an already existing IR *X* of a neighboring domain and reuse the available lowerings (e.g., IR X → IR Y). Finally, to fully unleash the potential for flexibility and extensibility, we propose to extend the scope of query compilers to also perform *query optimization as a sequence of compiler passes*. This enables interleaving query optimization passes with existing compiler passes, implementing new ideas for query optimization, or optimizing IRs of other domains. Thus, we can then perform effective cross-domain optimization in a natural way and speed up data processing by more than one order of magnitude [9]. Based on these ideas, we built a SQL-capable prototype system called LingoDB that is flexible and extensible yet reaches a competitive performance to state-of-the-art systems.

LingoDB is based on four new MLIR dialects that are designed as open IRs for data processing. Common query optimizations are now implemented as compiler passes. Lowering passes are used to lower the MLIR dialects until one final, low-level dialect is reached. Using LingoDB, we first show that our approach based on MLIR results in much lower compilation latencies compared to existing layered query compilers. Afterward, we compare the codebases of LingoDB, NoisePage, and DuckDB, to show that by using MLIR and by implementing query optimization as compiler passes we can significantly reduce the implementation effort. Finally, we show the extensibility of our approach by using a case study that integrates a PyTorch model into a SQL query, thereby demonstrating the opportunities of cross-domain optimizations combined with query optimization.

In this work, we first motivate the need for an open query compilation stack in Section 2 and give an overview of MLIR in Section 3. Building on prior work that we revisit in Section 4.1, we derive a compilation stack using MLIR that uses multiple dialects, implements query optimization as compiler passes, and overcomes previous obstacles of layered compilation in Section 4. In Section 5 we describe four new MLIR dialects for data processing, as well as their corresponding lowerings and optimizations. Afterward, we explain the design of LingoDB (cf. Section 6) before evaluating its performance in Section 7. Finally, we discuss software economics and extensibility in Sections 8 and 9 before concluding by outlining the future work potential in Section 10.

2 THE NEED FOR AN OPEN QUERY COMPILATION STACK

Over the last decade, dynamic code generation has continuously pushed the limits of SQL analytics in academia [5, 10, 23], and industry [7, 33, 35]. Today, many compiling database systems use data-centric code generation [22] to translate SQL queries to efficient machine code. Despite these advances, code generation still presents a major challenge for database developers as we have not yet reached a consensus on how a full query compiler should look like [11, 13, 32, 34]. Especially, no *open* query compilation stack exists. For us, an *open* stack for query compilation should fulfill three major aspects:

Openly Available. An openly available query compilation stack would significantly reduce the entry barrier for research on compiling databases. Today, without such a framework, building a compiling query engine is a non-trivial undertaking. It requires implementing a suitable type- and runtime system, code generation for relational operators, and efficient low-level frameworks like LLVM to yield low compilation latencies. Additionally, compiling query engines require novel solutions for long-solved tasks such as profiling and debugging since input query and generated code are only loosely coupled [3, 12].

Open with regard to flexiblity. The absence of a common denominator diversifies the development and makes many systems incompatible and incomparable to one another. This makes it hard to benefit from the latest research [2, 4, 6, 11, 15, 26, 27] in this area as new advances are often tied to architectural details of the present code generation stack and cannot be easily implemented in one's system without also adopting their original system architecture.

Open with regard to Extensibility. Extensibility has not been an important design goal for many query compilers. Since their design is focused on relational algebra, integrating other high-level concepts like machine learning operations is non-trivial. This is unfortunate as it fails to support cross-domain optimizations that were shown to be very effective [9]. Extending the support to new, increasingly popular processing domains like UDFs, machine learning, or graph processing is unnecessarily hard. Combined with the lack of flexibility, we cannot even benefit from the fact that these other processing domains have already built their own domain-specific compilation stacks.

We argue that research around compiling database systems lacks an *open* platform where ideas can flourish and grow. In the compiler community, this role is taken on by LLVM, which has established itself as the shared environment for novel compiler optimizations. An equivalent framework for compiling database systems would spare researchers the reimplementation of fundamentals and reduce the entry barrier for new ideas. In this work, we propose a platform that achieves abstraction, ease of development, flexibility, and extensibility to other domains without sacrificing performance.

3 BACKGROUND: MLIR

Building high-level domain-specific compilers is a complex task, and the resulting codebases are often hard to extend and maintain. The developers of Tensorflow [1] made similar observations as modern machine learning frameworks are composed of various representations (e.g., Tensorflow graphs and XLA HLO), use many

different compilers (e.g., TensorRT, XLA), and target a multitude of runtime systems (e.g., CPU, TPU). They, therefore, initiated MLIR, short for *Multi-Level Intermediate Representation*, which is a new flexible and extensible compiler infrastructure. It aims to reduce software fragmentation, support compilation for heterogeneous hardware out of the box, and reduce the cost of developing domain-specific compilers through the reuse of common infrastructure and existing intermediate representations.

Existing compiler frameworks like LLVM are successful because they enable and simplify the reuse of common compiler technologies. However, LLVM (and similar systems) only offer a single abstraction level. In the case of LLVM, this is a low-level, instruction-based IR. This single abstraction level might be too low or too high for certain analysis and optimization. Thus, many programming languages like Swift, Rust, or Julia invent their own language-specific intermediate representations on top of, e.g., LLVM to perform high-level, language-specific optimizations. Still, developing and maintaining custom IRs is expensive since one has to implement and maintain common functionality for e.g., debugging and diagnostics. MLIR aims to solve this problem by providing an infrastructure for introducing new abstraction levels in the form of IR dialects.

```
%results:2 = d.operation(%arg0, %arg1) ({
  // Regions are attached to operations          Region
  // and can contain multiple blocks:
  ^block(%argument: !d.type):                    Block
    // nested operations use block arguments:
    %calculated = d.calc %argument : !d.type
    d.return %calculated : !d.type
})
// Ops can have a list of attributes:
{attribute="value" : !d.type} : !d.type,!d.type
```

Figure 2: MLIR's IR format – *An operation can take typed SSA-values as operands and produce result values. Furthermore, regions with blocks can be attached. Each block can take block arguments that are processed by nested operations.*

3.1 Intermediate Representation

MLIR uses a single, standardized intermediate representation. Custom abstraction levels are introduced as *MLIR dialects* that conceptually group a set of *types* and *operations*. This unified intermediate representation uses the Static Single Assignment form (SSA) [21], which simplifies dataflow analysis and is widely used in the compiler community. Unlike many SSA-based IRs, MLIR uses nested regions instead of a flat control flow graph to natively support high-level abstractions like loops. Figure 2 shows an example operation with one region that contains nested operations. In MLIR, SSA-values always have an associated type that is specified by a dialect. MLIR further allows annotating operations with named attributes to represent compile-time information like constants. Beyond modeling constants, they are also useful for lowering hints, debugging information, and optimization. Additionally, the authors of MLIR argue that the original source location and applied transformations should be easily traceable for operations. Thus, each operation stores source location information in a dedicated MLIR attribute. This detailed source code tracking simplifies debugging as well as profiling across code transformations.

MLIR dialects can be combined in order to leverage built-in dialects and preserve high-level operations as long as necessary. This also allows to progressively lower programs to the hardware abstraction level in small, isolated steps. To simplify these lowerings, MLIR provides infrastructure for pattern-based rewrites.

3.2 Built-in Dialects

MLIR makes it easy to define new, custom dialects but also comes with several built-in dialects that can be reused. These dialects can be split into three groups. First, high-level dialects that are domain-specific and, e.g., provide linear algebra operations. Second, are the so-called mid-level dialects that abstract from hardware details but are mostly general-purpose (e.g., `arith` that provides arithmetic operations). Third are low-level dialects that are backend-specific (e.g., `llvm`). In this section, we will mainly discuss the mid-level and low-level dialects, on top of which we build custom dialects.

The following five dialects are among MLIR's mid-level dialects: A central `builtin` dialect defines common types like integers, floating-point, or function types as well as the corresponding attributes. Additionally, it also provides `module` and `func` operations to represent modules and functions. On top of the `builtin` dialect, the so-called `std` dialect adds basic primitives for (conditional) branching, function calls, and atomic memory operations. Common arithmetic and logic operations on integers and floating-point values are provided in a separate `arith` dialect. Finally, operations for creating and manipulating memory references are defined by the `memref` and high-level control-flow operations like `for`, `while`, and `if` by the `scf` dialect.

MLIR also comes with a variety of *backend* dialects that represent different targets. This includes a `llvm` dialect that represents a useful subset of the LLVM IR, a generic `gpu` dialect, and hardware specific dialects such as a `nvvm` dialect for CUDA or a SPIR-V dialect (`spv`). However, high-level dialects can usually lower to mid-level dialects without defining lowerings to low-level dialects like `llvm`.

3.3 Passes

In addition to its built-in dialects, MLIR also provides its own compiler passes. They range from dialect-specific passes to generic passes that can transform arbitrary dialects. The generic *inlining* pass, for example inlines *callable* operations such as functions. Additionally, MLIR can perform generic *common subexpression elimination (CSE)* that relies on traits to specify side-effects. And finally, MLIR provides a *canonicalization* pass that performs dead code elimination, constant folding, and constant hoisting and applies operation-specific rewrite patterns.

4 DESIGNING AN *OPEN* QUERY COMPILATION STACK BASED ON MLIR

As noted by Tahboub et al [34], database systems are essentially domain-specific compilers. In conventional systems, the journey of a SQL query, as depicted on the left side of Figure 1, usually starts as input for a parser, very similar to the compilation of high-level programming languages. The output of the SQL parser is an abstract syntax tree (AST) that is transformed into a logical plan. The

database then analyzes the semantics of this plan and optimizes it using well-known techniques like predicate pushdown, query unnesting, and join order optimization. Afterward, the optimizer transforms the logical plan into a physical plan by selecting a specific implementation for every node. If the database system employs query compilation, this physical plan serves as input for the query compiler that emits a compact query program. In this last step, databases usually leverage compiler frameworks like LLVM.

4.1 Prior Work

Over a decade ago, HyPer pioneered data-centric code generation [22] to generate tight loops across operator borders while avoiding materialization whenever possible. Using LLVM for code generation enabled high performance and low compilation latencies in the two-digit millisecond range. Kohn et al. further improved compilation latencies by introducing Adaptive Execution [14]. However, HyPer's approach for code generation was contrasted by Klonatos et al. for being too low-level and missing optimization potential [13]. Instead, they proposed to use high-level languages like Scala for building query compilers and demonstrated this with LegoBase.

In 2016, Shaikka et al. [32] proposed building layered query compilers as an alternative to complex, monolithic query compilers that are hard to extend and maintain. As shown in Figure 1, an optimized, physical plan is translated into a first, high-level declarative intermediate representation. This *IR 1* is then lowered progressively over several intermediate representations until a low-level *IR N* is reached. This final IR is then translated to, for example, a C program to obtain executable machine code.

Such a layered approach has multiple advantages: First, query compilers get more flexible. The effort to support a new frontend is reduced to lowering into an existing IR in the stack. Conversely, supporting a new backend only requires translating an existing IR at a lower level to machine code. Second, new medium layers of abstraction facilitate new optimizations for which previous abstraction levels were either too low or too high. Finally, splitting the lowering process into multiple steps also reduces the overall complexity (separation of concerns). However, these advantages come at the cost of increased compilation times and having to implement multiple IRs and compiler passes, which can require much effort depending on the available infrastructure. Alternatively, Tahboub et al. propose using the *first Futamura projection* [34] to generate query compilers from query interpreters. This reduces the overall complexity but also decreases flexibility and extensibility.

In 2020, Müller et al. proposed the Collection Virtual Machine (CVM), a common compiler infrastructure for collection-oriented IRs on multiple abstraction levels [21]. Its goals are similar to MLIR's, but CVM is constrained to collection-oriented IRs, which limits the scope. With MLIR, we can represent high-level collection-oriented IRs as well as scalar IRs at lower levels. Furthermore, unlike MLIR, CVM is not publicly available, which hinders any collective effort to develop a broad spectrum of IRs for specialized use cases.

4.2 Practical Layered Query Compilation

As we discussed above, a layered approach to query compilation offers high flexibility and extensibility. However, it also requires implementing multiple layers of IR and compiler passes and often comes with high compilation latencies. In this work, we show that MLIR makes layered query compilation more practical. First, MLIR can significantly reduce the effort for building a stack of IRs significantly by offering declarative syntax for specifying IRs and rewriting patterns. For example, MLIR allows reusing entire dialects to abstract from control flow and use built-in compiler passes. Second, MLIR can decrease compilation times significantly compared to prior work: It was written from scratch in C++ and is optimized for efficient, layered compilation. For example, types and attributes are represented as *uniquified* C++ objects, i.e., for each type with the same type parameters, exactly one C++ object is created. Furthermore, MLIR is also designed to optimize and compile larger programs multi-threaded. Finally, existing prototypes like DBLab [32] or Legobase [13] usually emit a standalone C program that is separately compiled, which causes very high compilation latencies. In contrast, we use LLVM for machine-code generation and executed code can call arbitrary library functions of the current process. This enables competitive compilation latencies since we do not have to compile the complete runtime per query. Furthermore, generated code can also call into a precompiled, shared runtime library, making it feasible to implement a complete database system around the compiling query engine.

4.3 Open IRs

Shaikka et al. proposed a layered approach with one IR per layer and constrained the lowering process by requiring a unique lowering path between two layers. In this work, we adapt this approach and propose using multiple *open* IRs per layer to achieve higher reusability and extensibility. This allows for high extensibility as one can combine arbitrary IRs at one level, e.g., use IRs representing UDFs or other forms of analytics within relational operators. This makes query compilation much more powerful as one can reuse and recombine already existing IRs. To fully profit from reusability and recombinability, we design our database-related IRs to work seamlessly with other IRs. For example, a join operator can use arbitrary IRs as part of its predicate by attaching the predicate as an MLIR region. As sketched in Figure 1, lowering also differs from previous approaches: We first translate SQL (combined with UDFS, machine learning, ...) into a combination of IRs that e.g., represent relational operators and scalar expressions. We can then lower these IRs independently from each other, e.g., first lower relational operators (IR1) into control-flow (IR3), while keeping the scalar expressions (IR2) used in predicates or computations. This also allows keeping *embedded IRs* as long as we want without implementing corresponding operations at each level.

4.4 Integrating Query Optimization

Prior work [32, 34] assumes that query optimization is performed *before* invoking the query compiler. This reduces the complexity of query compilers but comes at a high cost.

First, query optimizers require a lot of infrastructure that is already available in compiler infrastructure, for example, to match patterns, rewrite rules, and perform fixpoint iterations. Even whole optimization passes like common subexpression elimination are already performed by compilers. Second, relational workloads are

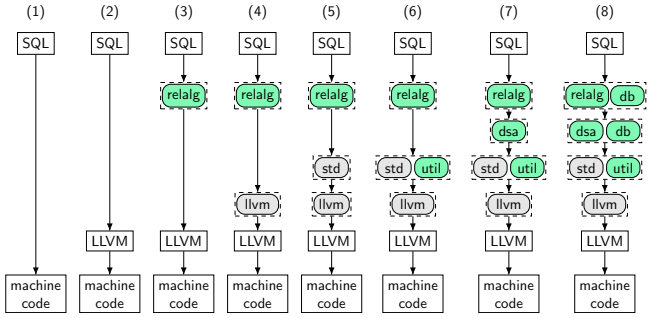

Figure 3: Visualization of the iterative design process

increasingly entangled with external code for machine-learning or complex UDFs. However, performing cross-domain optimization within a classical query optimizer is hard, despite the huge potential. Logical plans within databases are structured as trees of relational algebra operators and differ significantly from the control flow of non-relational data processing. Cross-domain optimization has to interleave query optimization techniques with control-flow analysis for optimal results.

To make cross-domain optimization practical, we have to close the gap between query optimizers and compilers. This can either be achieved by extending the query optimizer with compiler techniques to also reason about e.g., control-flow or convert query optimization into a sequence of compiler passes that can be interleaved with other domain-specific passes. Since we already use compiler infrastructure for query compilation, it is only natural to choose the latter and move the whole query optimization into the query compilation phase.

4.5 Open Query Compilation Stack using MLIR

Similar to Shaikka et al., we derive our MLIR-based compilation stack in seven steps as sketched in Figure 3.

(1) Overall, we want to compile SQL queries into machine code;
(2) We use LLVM for generating optimized machine code
(3) However, SQL is not suitable for query optimization; Therefore, we introduce a *relational* MLIR dialect into which SQL can be translated;
(4) To reduce the overall complexity, we reuse the existing `llvm` dialect instead of directly translating to LLVM IR;
(5) We want to work on a higher abstraction level than llvm, both to reduce effort and increase flexibility; Thus, we reuse existing *standard* dialects on top of the `llvm` dialect;
(6) However, *standard* dialects miss capabilities like e.g., working with composite types; We thus add a `util` dialect which provides such features independent of the used backend dialect to increase flexibility;
(7) To abstract from implementation details of algorithms and data structures, we introduce a `dsa` dialect into which relational operators can be lowered;
(8) To avoid reimplementing database-specific scalar operations that can e.g., handle null values on multiple layers, we introduce a separate `db` dialect that is used for specifying e.g. predicates but is still available after lowering relational operators.

```
module{
 func @main () -> !db.table {
   %1 = relalg.basetable {a=>@R::@a({type=!db.nullable<i64>})}
   %2 = relalg.basetable {b=>@S::@b({type=i64})}
   %3 = relalg.join %1, %2 (%4: !relalg.tuple) {
     %5 = relalg.getcol %4 @R::@a : !db.nullable<i64>
     %6 = relalg.getcol %4 @S::@b : i64
     %7 = db.compare %5, %6
     relalg.return %7 : !db.nullable<i1>
   }
   %4 = relalg.materialize %3 [@R::@a ,@S::@b ]
                           => ["R.a","S.b"] : !db.table
   return %4 : !db.table
 }
}
```

Figure 4: A simple query (`select R.a, S.b from R,S where a=b`) is represented by operations of different MLIR dialects.

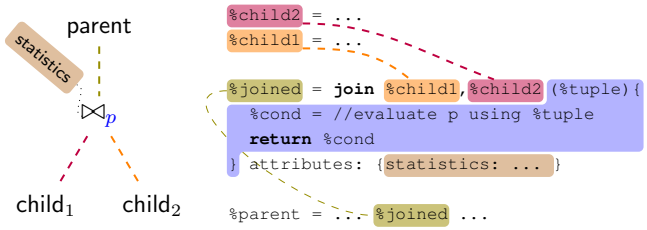

Figure 5: Representing plan nodes in MLIR – *Child and parent relationships are expressed through A-values. User-defined expressions such as join predicates are represented as single-block regions. Statistics and her metadata are attached as attributes.*

5 DIALECTS FOR DATA PROCESSING

In the following section, we explain the design of the four new MLIR dialects (`relalg`, `db`, `dsa`, and `util`) as discussed in Section 4.5. To give an outlook, Figure 4 illustrates how a simple SQL query can be represented using a combination of different MLIR dialects: The relational `relalg` dialect provides relational operators while predicates are represented by operations of the `db` dialect. Modules and functions are declared using built-in operations.

5.1 `relalg`: Relational Algebra

For database engines, relational algebra provides the theoretical foundation for query processing. While classical relational algebra is specified on sets of tuples, logical operators in databases usually work on tuple streams as this enables ordering and multi-set semantics. Thus, we propose a high-level declarative MLIR dialect with relational operators that produce and/or consume values of type `!relalg.tuplestream`. However, in contrast to other intermediate representations, this dialect only includes relational operators, but not operations for building expressions used in e.g., predicates. Expressions can be freely constructed using arbitrary MLIR operations.

In addition to well-known operators like selections, maps, and joins we also include three additional join operators required for unnesting correlated subqueries as proposed in [24]: A `singlejoin` corresponds to a left outer join but expects exactly one tuple on the right side. A `markjoin` adds a new column to the left side that indicates if any tuple on the right fulfilled the join conditions. Similarly,

for each tuple on the left side, a `collectionjoin` materializes all matching tuples on the right side and appends the resulting list as a column. Furthermore, we also add operations for materialization and working with tuples and column values. Altogether, the types and operations listed in Table 1 enable formulating and optimizing complex, relational queries.

Since columns (i.e., relational attributes) are one of the core ingredients of relational algebra, we need to model them in MLIR. In order to refer to columns declaratively, we represent them as MLIR attributes similar to the way symbols are implemented in MLIR. We, therefore, introduce two new attributes: One for declaring a new column (e.g., during a `map` operator), and one for referring to a previously declared column (e.g., inside a `selection` operator, or the grouping specification for an `aggregation`). To simplify the usage we pretty print both attributes: `@scope::@col(type:!type)` declares a new column `@col` in the namespace `@scope` with the provided type, and `@scope::@col` refers to it.

Many complex relational operators contain expressions either as conditions (selections and joins) or for computing new values (e.g., map, aggregation). As already discussed in Section 4 and shown in Figure 5, we represent such expressions as attached MLIR regions with nested, arbitrary operations. The current tuple of type `!relalg.tuple` is provided as an argument for the region, and one can access the current value for specific columns using the `getcol` operation on the tuple value. To simplify analysis and make it more generic, operators implement different *MLIR Interfaces* that allow calling interface methods (e.g. `getAvailableColumns()`) independent of the concrete operation. Metadata such as statistics or estimates can be attached as MLIR Attributes. Overall, using the design decisions outlined above, we can map all concepts of classical plan nodes to MLIR as shown in Figure 5.

Table 1: Types and Operations of the `relalg` dialect

	Type / Operation	Description
types	tuple	conceptual tuple of relational algebra
	tuplestream	stream of tuples produced by operators
relational operators	`const_relation`	creates a stream from constant tuples
	`basetable`	returns a tuple stream from a scanned table
	`selection`	$res := \sigma_{[region]}(arg_0)$
	`map`	$res := \chi_{[region]}(arg_0)$
	`projection`	$res := \Pi_A(arg_0)$ (distinct possible)
	`renaming`	$res := \rho_{@b \leftarrow @a,\ldots}(arg_0)$
	`aggregation`	$res := \Gamma_{[attributes];[region]}(arg_0)$
	`crossproduct`	$res := arg_0 \times arg_1$
	`innerjoin`	$res := arg_0 \bowtie_{[region]} arg_1$
	`semijoin`	$res := arg_0 \ltimes_{[region]} arg_1$
	`antisemijoin`	$res := arg_0 \triangleright_{[region]} arg_1$
	`outerjoin`	$res := arg_0 ⟕_{[region]} arg_1$
	`singlejoin`	$res := arg_0 ⟕^{1}_{[region]} arg_1$
	`markjoin`	$res := arg_0 ⟕^{M:m}_{[region]} arg_1$
	`collectionjoin`	$res := arg_0 ⟕^{C:c}_{[region]} arg_1$
	`sort`	sorts the tuple stream by some attributes
	`limit`	only return the first #num tuples of the stream
	`tmp`	caches a tuple stream to avoid reevaluation
tuple support	`getcol`	$res := arg_0.@col$
	`materialize`	$res := astable([columns], arg_0)$
	`getscalar`	$res := aslist(column, arg_0)[0]$
	`getlist`	$res := aslist([columns], arg_0)$
	`exists`	$res := \lvert arg_0 \rvert > 0$
	`in`	$res := \exists t \in arg_0 : t.[@a] = arg_1$

5.2 Query Optimization Passes

As motivated in Section 4, we implement standard query optimization techniques commonly performed on plan nodes as compiler passes over MLIR operations. In this section, we describe how we implemented five common query optimization techniques in MLIR as sketched in Figure 6: ① expression simplification, ② query unnesting, ③ selection pushdown, ④ join order optimization, and ⑤ physical optimization.

First, we try to simplify the query expressed as an MLIR module as far as possible. Thus, we apply MLIR's built-in `CSE` pass to eliminate common subexpressions and the `canonicalization` pass to eliminate dead-code and to apply canonicalization patterns. To perform database-specific simplifications, we add corresponding canonicalization patterns which, e.g., extract common conditions from disjunctive expressions. By performing these simplifications first, we enable optimizations like pushing predicates further down.

After simplification, we perform query unnesting in three steps : First, nested `relalg` operations i.e., a subquery inside a `selection` operation are extracted. Here, it is crucial to correctly handle values that are defined in the surrounding operator but are used inside a region of the nested operation. Second, implicit joins represented by `getscalar`, `getlist`, `exists`, or `in` operations are transformed into explicit joins represented by `singlejoin`, `collectionjoin`, and `markjoin` operations. Finally, we apply the unnesting rules proposed by Neumann et al. [24] and restructure the operations accordingly.

Next, we push selections down in two steps. We first split selections with a conjunctive predicate as well as `map` operations that compute more than one attribute. For this, we need to conceptually split the corresponding regions as well, which might require duplicating operations. Afterward, a second pass pushes selections down as far as possible and moves the `selection` operation before its new child.

After selections have been pushed down, we optimize the join order. We first scan the MLIR module for operations that materialize a tuple stream or cannot be handled by algorithms for optimizing the join order. For each of these operations, we then optimize the join-order of the corresponding subtree: First, we build a query graph that references the involved operations and annotate cardinality and selectivity estimations that are attached as MLIR Attributes to the `basetable` operations. On this query graph, we then apply dynamic programming [20] to yield the best join plan, which is then used to reorganize and restructure the already existing MLIR operations.

Finally, we perform physical optimization by introducing `tmp` operations to temporarily materialize a tuple stream consumed multiple times and deciding on an implementation for each operator. The implementation choice is then annotated as MLIR attribute for every operator and respected during lowering.

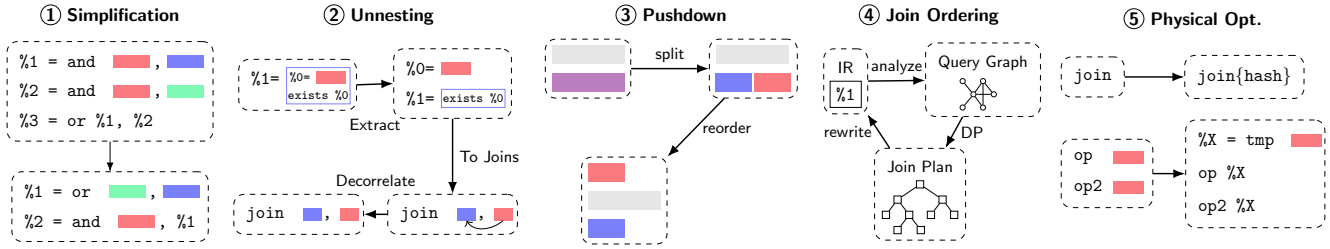

Figure 6: Visualization of different steps of query optimization

5.3 Lowering of Relational Operators

After running the custom compiler passes, we lower the declarative `relalg` dialect into a combination of imperative dialects. We use data-centric code generation as proposed by Neumann et al. [22] to transform declarative to imperative operations, according to annotations introduced during query optimization. During the lowering pass, we scan the MLIR module for operations that materialize a tuple stream. For each materializing operation, we traverse the corresponding operator tree represented by `relalg` operations. During the traversal, we build a tree of translator objects that implement `produce()` and `consume()` methods according to the producer-consumer model used by data-centric code generation. Calling `produce()` on the top-level translator object then inserts new imperative db operations that implement the operator tree. After every materializing operation is processed and removed, we run MLIR's `canonicalize` pass to clean up now unused declarative operations and simplify the lowered module. Figure 7 shows this lowering into imperative operations for an example query. Note how the highlighted predicate does not change besides lowering the `getcol` operation.

5.4 db: Database-specific Types and Operations

Table 2: Types and Operations of the db dialect

	Type / Operation	Description
types	`decimal<p,s>`	decimal with precision p and scale of s digits
	`char<len>`	fixed-length string type with length *len*
	`string`	variable-length string type
	`date<unit>`	date type with different units
	`interval<unit>`	interval type with different units
	`timestamp<unit>`	timestamp type with different units
	`nullable<type>`	nullable version of *type*
null	`null`	creates null value
	`isnull`	returns true if value is null
	`nullable_get_val`	return scalar value for nullable
	`as_nullable`	create a nullable value from a scalar value
scalar operations	`constant`	$res := c$
	`add, sub, mul, div, mod`	arithmetic operations with null semantic
	`compare,`	$res := arg_0\ pred_{\text{null}}\ arg_1$
	`between, oneof`	compare value with range or set of values
	`and, or, not`	logical operations with null semantic
	`cast`	semantic cast (e.g., string → int)
	`hash`	compute a 64-bit hash for a value
	`runtime_call`	$res :=$ "$Func$"$(arg_0, arg_1, ...)$

```
%0=relalg.basetable {a=>@R::@a(type=!db.string),b=>...}
%1 = relalg.selection %0 (%arg0: !relalg.tuple){
  %a = relalg.getcol %arg0 @R::@a
  %s = db.constant("STR")
  %cmp = db.compare eq %a, %s
  relalg.return %cmp : i1
}
%table = relalg.materialize %1 [@R::@b] => ["b"]
```

(a) Simple Query expressed as MLIR Operations

```
%tb = dsa.create ["b"] -> !dsa.table_builder
%scan = dsa.scan_source "{...}"
dsa.for %chunk in %scan  {
  dsa.for %row in %chunk {
    %a = dsa.at %row[0] : !db.string
    %s = db.constant("STR")
    %cmp = db.compare eq %a, %s
    %5 = db.not %cmp : i1
    dsa.cond_skip(%5 : i1)
    %b = dsa.at %row[1] : i32
    dsa.append %tb, %b
    dsa.next_row %tb
  }
}
%table = db.finalize %tb -> !db.table
```

(b) Same Query after lowering to imperative operations

Figure 7: Lowering a query from `relalg` to db

In addition to the `relalg` dialect, we implemented a second imperative db dialect with database-specific types and operations. These can then be used to represent common expressions in SQL queries as well as implement e.g., comparisons that get created during the lowering of an aggregation.

The type system of database systems usually deviates from the built-in types of most programming languages as they support types for storing dates, timestamps, or fixed-point decimal numbers. Since MLIR is mainly focused on numerical application, it does not offer any scalar types beyond integer and floating-point types. Hence, the db dialect adds 6 scalar types as shown in Table 2.

Furthermore, relational databases have the notion of null values that come with their own semantics. To reduce the overhead and only handle null values where necessary, we introduce an explicit nullable type that specifies that a value can also be null and add four supporting operations. For example, a value of type `!db.nullable<i32>` can either be an integer value or `null`.

On top of the available types (bool (1-bit integer), integers, floating-point, database-specific, and nullable types), we define operations

for defining constants, performing basic arithmetic and logical operations, comparing, hashing, and casting them. For operations beyond this basic set, we provide a `runtime_call` operation that calls a registered runtime function (e.g. "ExtractYearFromDate"). This allows for adding new functionality without new operations.

We implemented one optimization pass (`-db-eliminate-nulls` that transforms many operations on nullable types into operations on non-nullable types. This can be achieved by introducing explicit null checks and control-flow if necessary. After this simplification step, one can perform a lowering pass that transforms types and operations of the `db` dialect into a combination of `arith`, `std`, `scf`, and `util` types and operations. For this, we provide a set of MLIR rewrite patterns and specify the type conversion. Everything else is handled by the MLIR infrastructure.

5.5 `dsa`: Data Structures and Algorithms

Besides operations on scalar types, database engines also have to build and process a variety of complex data structures. This includes database tables, buffers for materialization, but also (temporary) index structures like hashtables.

Therefore, the `dsa` dialect provides a generic `for` operation that can iterate over a variety of iterable types (Figure 7b, line 3 and 4). The `cond_skip` operation can skip the remainder of the current iteration and proceed with the next value based on a boolean value (Figure 7b, line 9). This is especially useful for efficiently evaluating conjunctive predicates.

Figure 7b also shows an example for building a result table: Initially, we create a `table_builder` in line 1. For each matching tuple, we append the corresponding value in line 11 using the generic `append` operation and switch to the next row using the `next_row` operation. Finally, we use the `finalize` operation in line 16 to yield the final Apache Arrow table. In addition to the table builder, we also implemented a vector type and different hashtable types.

5.6 `util`: Utility Types and Operations

Since MLIR mainly focuses on numerical applications, many types and instructions that are critical for data processing tasks are not provided by built-in generic MLIR dialects. Up until now, only very low-level target-specific dialects like the `llvm` dialect offer these capabilities. To keep flexibility, reusability, and maintainability, we do not use the `llvm` dialect directly. Instead, we define a `util` dialect that provides the missing types and operations in an abstract way and implement a lowering to the `llvm` dialect using MLIR rewrite patterns.

Even though MLIR defines a `TupleType` in its `builtin` dialect, it does not provide operations to pack values into a tuple and unpack it later. We therefore add `pack` and `unpack` operations that operate on the already defined `TupleType`. Additionally, when dealing with a separate runtime system, we often need to provide the size of a type, e.g., to sort a buffer of values. While the size is still clear for single integers, it gets more complex with composite and custom types. The `util` dialect, therefore, provides a `sizeof` operation that also considers alignment and padding.

For data-processing applications, referencing data in memory and performing load and store operations is critical. MLIR already comes with a `memref` type and the corresponding operations. However, their design is focused on numerical applications like multiplying large matrices. To optimize for such cases, a value of the `memref` type holds not only a pointer but also metadata. While this overhead is not relevant when working with large matrices, even for small queries, it is noticeable during runtime but, worse, also impacts compilation times significantly. Since the built-in `memref` type is not suitable for simple and dynamically-sized references, we provide a simplified `ref` type for simple references and a `varlen` type for referencing variable-length data. These types are accompanied by a narrow set of operations that work on them (e.g., load a value from a typed reference).

6 SYSTEM OVERVIEW

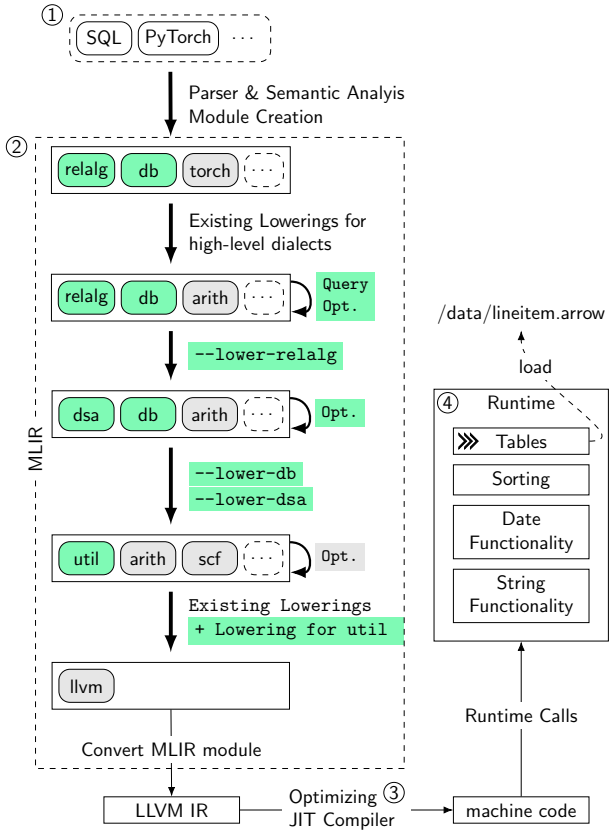

Figure 8: The developed prototype system consisting of ① a SQL frontend that performs parsing and semantic analysis, ② MLIR dialects, optimizations, and lowerings, ③ a LLVM-based optimizing JIT-compiler, and ④ a runtime system based on Apache Arrow

To evaluate the ideas proposed in Section 4, we implemented a prototype system called LingoDB based on the MLIR dialects explained in Section 5. Figure 8 gives an overview of LingoDB. It consists of four main components: ① Parsing and Semantic Analysis, ② MLIR-based optimization and lowerings, ③ LLVM-based jit-compilation, and ④ a dynamic runtime.

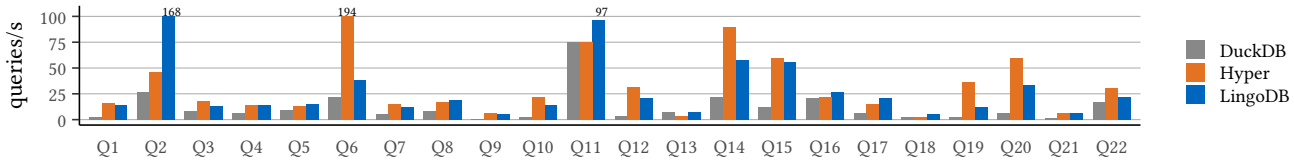

Figure 9: Query execution performance (compilation not included) for DuckDB, Hyper and LingoDB (SF=1)

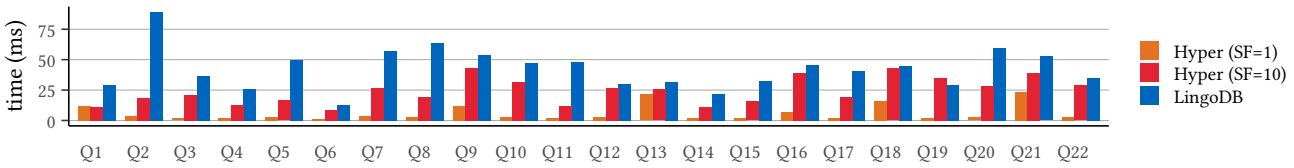

Figure 10: Total compilation times for Hyper and LingoDB.

In the first step, a provided SQL query is parsed and transformed into an equivalent representation in MLIR. The resulting MLIR module mainly consists of a relational `relalg` dialect and an imperative db dialect. As these dialects are statically typed, translating an untyped SQL query requires extensive semantic analysis to infer types.

The resulting MLIR module is then optimized by several MLIR passes that perform logical and physical query optimization. Afterward, high-level dialects are progressively lowered until the `llvm` dialect is reached. On each layer, we optimize the MLIR module using the canonicalization pass and specific passes for different dialects: For the db dialect, we eliminate null-handling and optimize used runtime functions to e.g., make the evaluation of constant like expressions efficient. When the standard control flow dialect is reached, we perform passes to move loop-invariant operations outside loops and move operations after checking conditions.

Then, one has to convert the MLIR module containing only `llvm` operations into an LLVM IR module and compile it to machine code. MLIR already provides an `ExecutionEngine` class that converts MLIR modules, performs LLVM optimizations, and generates machine code. As we emit already optimized LLVM IR due to MLIR passes, and compilation time is also a concern, we only perform four LLVM passes: `-instcombine`, `-reassociate`, `-gn`, and`-simplifycfg`. These passes perform basic peephole instructions, eliminate redundant instructions, and simplify the control flow. This set of LLVM passes is known to be useful for a wide variety of code [29].

Finally, we implemented a dynamically linked runtime system based on Apache Arrow. For a given database directory, the runtime loads all available files ending with ".arrow" as Apache Arrow tables into main memory. These tables are then identified by the basenames of the original file names. Using a set of runtime functions, one can then iterate efficiently over these tables in a batch-oriented manner. Additionally, the runtime system enables creating new Apache Arrow tables with a new schema constructed at runtime. This is especially useful to efficiently return query results back to the user. Complex functionality like sorting, string comparisons, or date arithmetic is also implemented in C++ and available through runtime functions.

7 PERFORMANCE EVALUATION

In this section, we evaluate the performance of LingoDB by looking into query execution and query compilation times. We then check how competitive our MLIR-based approach is compared to other solutions and how much overhead we introduce. Using the TPC-H benchmark, we compare our prototype system LingoDB with the vectorized database system DuckDB [31] and the commercial version of Hyper [10] published by Tableau [8].

All experiments were run single-threaded on a machine with an AMD Ryzen 9 5950X CPU with a base frequency of 3.4 GHz and a maximal frequency of 4.9 GHz. Each of the sixteen cores has an L1 data cache of 32 kiB, an L2 cache of 512 KiB, and shares an L3 cache of 64 MiB. The system runs Ubuntu 21.04 and provides 64 GiB of main memory.

In the first experiment, we compare the raw query execution times (no compilation time included) of Hyper, DuckDB, and LingoDB. To compensate for the fact that our prototype does not yet implement primary key indexes, we also do not create indexes for Hyper and DuckDB. Figure 9 shows the absolute query runtimes for running the TPC-H benchmark with a scale factor of 1.

On average, our prototype system is faster than DuckDB by a factor of 3.5. Also, for eight queries, we outperform Hyper because it is optimized for multi-threaded performance and therefore comes with a slight overhead when executed single-threaded. On the other hand, Hyper is faster for 12 queries due to advanced features like group joins, vectorized table scans, and early probing. Hence, Hyper is, on average 1.3 as fast as our prototype.

For compiling database engines, the raw execution time is only half of the truth as the query must first be compiled in contrast to interpreting engines. Thus, compilation times matter, especially for smaller datasets and short-running queries.

Existing layered query compilation prototypes written in high-level languages come with very high compilation latencies. For example, DBLab [32] reported more than 900 ms for TPC-H query 8 to internally lower and optimize intermediate representations. Additionally, CLang is invoked for generating machine code which takes another 300ms. In the remainder of this section, we show that

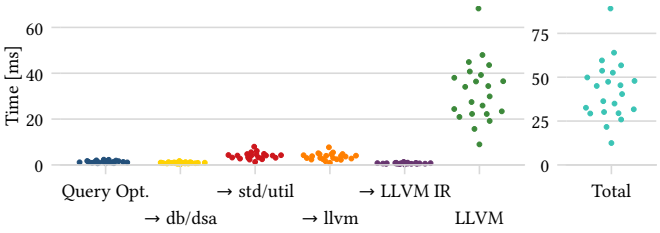

Figure 11: Times for different compilation phases of LingoDB – *Compilation phases: Query Optimization (Query Opt.), Lowering to imperative dialects using data-centric code generation (→ db/dsa), Lowering to standard dialects (→ std/util), Lowering to llvm dialect (→ llvm), Conversion to LLVM IR (→ LLVM IR), LLVM optimizations and machine code generation (LLVM)*

LingoDB compiles queries significantly faster, and thus, layered query compilation does not imply high compilation latencies.

Figure 11 shows the compilation times for all TPC-H queries and the different compilation steps. One important observation is that all the optimizations and lowerings in MLIR only take a few milliseconds, and most time is spent in LLVM for optimizations and machine-code generation. For example, the query that took the most time to compile (Q2) takes only 13 ms to optimize the MLIR module, lower it and convert it to LLVM IR. Then, LLVM takes 68 ms to optimize it and generate efficient machine code. This means that our layered approach to building a performant yet flexible and extensible query compiler does not bring any significant overhead.

Figure 10 compares the compilation times for Hyper and LingoDB for all TPC-H queries. Since Hyper is using Adaptive Execution [14] and avoids LLVM for short-running queries, we include Hyper both for scale factors 1 and 10. While we can observe that Hyper compiles significantly faster for SF=1, compilation times get much more comparable for SF=10 when Hyper also uses LLVM. However, even then, Hyper compiles faster than our prototype since LLVM is only used for computationally-intensive pipeline functions. On the contrary, our prototype compiles the whole query with LLVM, leading to higher compilation times. We note, however, that Adaptive Execution is an orthogonal optimization that can also be applied to LingoDB in the future.

8 DISCUSSION: SOFTWARE ECONOMICS

Developing a compiling query engine is a complex task that often results in a large, monolithic codebase. In this section, we discuss how building on MLIR reduces the complexity significantly. We, therefore, compare selected parts (that match the implemented features) of the codebase between LingoDB, DuckDB, and NoisePage, a system that employs query compilation and has reported similar performance to Hyper for TPC-H queries [19]. Numbers do not include blank lines, comments, or lines only containing brackets to account for different code styles.

Query Optimization. Since we propose integrating query optimization into the query compiler, we first compare the effort to implement query optimization in the three systems. As we can observe in Figure 12, LingoDB requires less than 2000 lines of code, whereas DuckDB requires more than 3x and NoisePage more than 5x as much code. Two main reasons cause this: First, both NoisePage and DuckDB need to reimplement optimizations like CSE that MLIR already provides. Second, NoisePage and DuckDB have to build an optimization infrastructure from scratch that can apply rewrite patterns. On the contrary, MLIR already provides an efficient infrastructure for which one only has to provide optimization and canonicalization patterns.

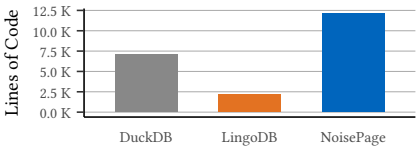

Figure 12: Lines of Code for Implementing Query Optimization in LingoDB, NoisePage, and DuckDB

Query Execution. Next, we compare the three systems regarding code complexity used for executing an already optimized query. Even though the three different systems use different strategies for executing queries, we can split the corresponding codebases into four categories:

(1) **IR**: This includes all code required for implementing different intermediate representations (including the optimized plan).
(2) **OperatorImpl**: This includes all code required for implementing the relational operators, either in a vectorized way or by emitting a lower-level IR.
(3) **Lowering+Backend**: This includes all further lowering steps for imperative IRs as well as implementing an execution backend via e.g., LLVM.
(4) **Runtime**: All runtime functionality that is implemented in C++ and required by the query execution layer.

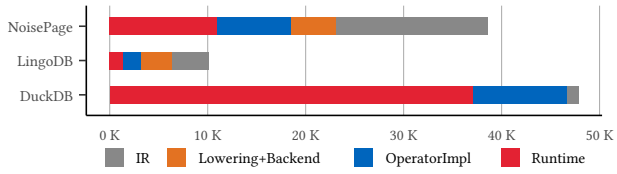

Figure 13: Lines of Code for Implementing Query Execution in LingoDB, NoisePage, and DuckDB

Figure 13 displays the code complexity for the three systems and the four categories. Overall, we can observe that our prototype requires about 3x less code than NoisePage and 4x less code than DuckDB for implementing a similar feature set.

Since DuckDB executes queries vectorized, it does not need other intermediate representations than the optimized plan. However, vectorized execution must provide one function for every operation, type, and underlying storage. This leads to a very large runtime system. In contrast, using query compilation makes an

extensive runtime unnecessary as we can generate specialized code. We, therefore, use runtime only where it makes sense to reduce the compilation time (by avoiding inlining large code snippets) or to interact with, e.g., the storage engine.

Overall, NoisePage's bar looks similar to the one of LingoDB as NoisePage also uses query compilation with three layers of abstraction: declarative query plan, imperative, high-level IR, and a bytecode format. However, compared to LingoDB, NoisePage requires much more code to implement these IRs because of three main reasons: First, NoisePage cannot reuse MLIR infrastructure and instead has to reimplement many components. Second, compared to LingoDB, NoisePage cannot build on existing IRs and has to define and implement lowerings for operations that MLIR already provides. Finally, MLIR allows to define entire dialects declaratively in a *TableDef* format from which MLIR's *TableGen* utility generates C++ code. This significantly reduces the effort for adding new dialects as, for our dialects, 1 thousand lines in the *TableDef* format correspond to around 20 thousand lines of generated C++ code.

Furthermore, the second difference is the size of the runtime. Whereas LingoDB generates machine code for many operations, NoisePage relies more on precompiled runtime functions to simplify the interpretation of the bytecode format at the cost of a more complex runtime.

Testing. For the development of compilers as well as database engines, testing is crucial to avoid new bugs. However, compilers and databases are usually tested slightly differently: Database engines are commonly tested through unit tests specified in the used programming language and by running end-to-end SQL tests. In contrast, it is quite common for compilers to have text-based tests. The compiler then processes the contained code fragment, and the expected behavior is checked against comments in the test file. This simplifies writing new tests and abstracts from internal representations. Building on top of LLVM, MLIR also uses text-based tests successfully for built-in dialects. Additionally, since MLIR enforces that every operation is printable and parseable, every dialect can be tested that way. For our prototype, we test all dialects and transformations, including query optimization, using text-based testing. This allows us to reach high test coverage without writing repetitive unit tests in C++. Furthermore, regression tests can be directly derived from a respective MLIR module by annotating the expected output as comments.

9 DISCUSSION: EXTENSIBILITY

As previous work [9, 26] shows, integrating different analytic frameworks into a single system can yield speedups of more than one order of magnitude. In the past, this was achieved by crafting custom IRs that covered different areas. With our approach, this is no longer necessary: we can focus on a relational dialect and mix it naturally with other domain-specific dialects.

In the following, we demonstrate this extensibility of our prototype by showing how we can extend LingoDB to also represent and cross-optimize a simple machine-learning model. In Figure 14a, a SQL query uses a trained PyTorch model for linear regression as a predicate. Traditional database systems can implement this query, if at all, by calling a user-defined function for each tuple. We, however, show how to leverage MLIR as a common denominator to cross-optimize and inline this model for efficient execution.

In previous sections, we already showed how SQL queries can be represented as MLIR operations of different dialects. Furthermore, using the Torch-MLIR project[30], we can translate the PyTorch model into corresponding MLIR operations of a `torch` dialect. Thus, both the query and the PyTorch model can be represented in one MLIR module with the query calling the model as depicted in Figure 14b. Note that this is made possible by designing relational operators to use MLIR regions for expressions. In a second step, we apply already existing lowerings and optimizations to transform `torch` operations into simple arithmetic operations as shown in Figure 14c. Finally, we perform query optimization. For our example, this means inlining the model function and expression simplification. Starting with an expression like $a * 2.06032777 + 0.581083298 < 5$, we can apply simple rewrite rules to finally yield the predicate $a < 2.144763938$ as shown in Figure 14d. Such a simple predicate can be executed efficiently and allows for more optimizations like pushing the predicate into the table scan to prune entire data blocks or efficient cardinality estimation using sampling.

In the future, we want to look into how to design transitions between different high-level dialects, how to generate mixed MLIR modules, and which cross-domain optimizations can be applied. This is relevant for properly integrating machine-learning frameworks that already provide MLIR dialects but can be extended to other areas like graph analytics using, for example, MLIR's SparseTensor dialect.

10 FUTURE WORK

The flexibility and extensibility of the proposed platform for building query compilers leave a lot of room for future work. Of course, we plan to make our prototype more feature complete and implement many optimizations regarding execution and compilation times. Additionally, we plan on working on the four areas discussed in this section.

Currently, our prototype runs single-threaded on a single machine and, therefore, cannot fully utilize modern hardware that is increasingly more parallel. Thus, in the future, we want to generate code for parallel and distributed systems. For implementing intra-query parallelism, we have two options: First, we could explicitly schedule pipeline functions with morsel-driven parallelism [18], which requires implementing a scheduler and adding corresponding MLIR operations. Alternatively, we could implement parallelism by only using already existing MLIR dialects such as the `omp` dialect that represents parallel execution using OpenMP [25].

While MLIR already supports parallel execution, it still lacks a dialect for modeling distributed systems. Thus, implementing distributed query processing in our prototype would require introducing new dialects. Similar to MLIR's `gpu` dialect, one could add a generic `distributed` dialect that provides high-level primitives. Further low-level dialects could then implement these primitives based on, for example, MPI or RDMA.

Supporting heterogeneous hardware is getting more and more important for database systems. Luckily, one of MLIR's design goals was to enable and simplify the compilation for heterogeneous hardware. Thus, we can benefit from the existing support without much

```
model = torch.nn.Linear(1, 1) // trained model
db.query("select a from R where model(a) < 5")
```

(a) A SQL query that uses a pytorch model as predicate

```
module{
  func @torch_model(%arg0: !torch.vtensor<[1,1],f32>){
    %0 = torch.vtensor.literal(dense<0.581083298>)
    %1 = torch.vtensor.literal(dense<2.06032777>)
    %2 = torch.aten.linear %arg0, %1, %0
    return %2 : !torch.vtensor<[1,1],f32>
  }
  func @model(%val : f32) -> f32 {
    %0=tensor.from_elements %val
    %1 = torch_c.from_builtin_tensor %0
    %2 = call @torch_model(%1)
    %3 = torch_c.to_builtin_tensor %2
    %c0 = arith.constant 0 : index
    %4 = tensor.extract %3[%c0,%c0]
    return %4 : f32
  }
  func @main () -> !db.table {
    %1 = relalg.basetable {a=>@R::@a({type=f32})}
    %2 = relalg.selection %1 (%3: !relalg.tuple) {
      %4 = relalg.getcol %3 @R::@a : f32
      %5 = call @model(%4) : f32 -> f32
      %6 = db.constant 5 : f32
      %7 = db.compare lt %5, %6
      relalg.return %7 : i1
    }
    %3 = relalg.materialize %2 [@R::@a ] => ["a"]
    return %3 : !db.table
  }
}
```

(b) MLIR module that represents both query and pytorch model

```
module{
  func @model(%val : f32) -> f32 {
    %cst = arith.constant 2.06032777 : f32
    %cst_0 = arith.constant 0.581083298 : f32
    %0 = arith.mulf %arg0, %cst : f32
    %1 = arith.addf %0, %cst_0 : f32
    return %1 : f32
  }
  func @main () -> !db.table {
    %1 = relalg.basetable {a=>@R::@a({type=f32})}
    %2 = relalg.selection %1 (%3: !relalg.tuple) {
      %4 = relalg.getcol %3 @R::@a : f32
      %5 = call @model(%4) : f32 -> f32
      %6 = db.constant 5 : f32
      %7 = db.compare lt %5, %6
      relalg.return %7 : i1
    }
    %3 = relalg.materialize %2 [@R::@a ] => ["a"]
    return %3 : !db.table
  }
}
```

(c) MLIR module after lowering the torch dialect

```
module{
  func @main () -> !db.table {
    %1 = relalg.basetable {a=>@R::@a({type=f32})}
    %2 = relalg.selection %1 (%3: !relalg.tuple) {
      %4 = relalg.getcol %3 @R::@a : f32
      %5 = arith.constant 2.144763938 : f32
      %6 = arith.cmpf olt %4, %5
      relalg.return %6 : i1
    }
    %3 = relalg.materialize %2 [@R::@a ] => ["a"]
    return %3 : !db.table
  }
}
```

(d) MLIR module after query optimization

Figure 14: Example: Integration of ML models

effort by relying on MLIR's standard dialects. For example, MLIR already offers built-in support for different GPUs, as GPU support is important for MLIR's main users: machine learning frameworks. Especially for application areas where queries are mostly static and compilation times do not matter, it could also be possible to leverage FPGAs since there are already related MLIR-based projects [28].

Over the last years, several intermediate representations have been proposed to improve query compilation. Previously, integrating such proposals was complex and labor-intensive. However, since MLIR significantly reduces the efforts needed to add new intermediate representations, we plan to integrate multiple proposals into the current prototype. At the moment, we directly lower the relational `relalg` dialect into the imperative `db` dialect. In the future, we want to add dialects in between that represent different kinds of sub-operators. Especially for the compilation of aggregations and window functions, this has been shown to reduce complexity and improve performance [15]. Other sub-operators could also reduce complexity through reuse and expose a non-relational, declarative interface [2]. Finally, low-level dialects that represent existing IRs like Umbra IR [11] could serve as additional backends with, e.g., faster compilation time.

11 CONCLUSION

This paper argues that research around compiling database systems lacks an open, flexible, and extensible platform. We propose building on MLIR, a new compiler infrastructure that aims to simplify the development of domain-specific compilers such as query compilers. First, we explained how to make layered query compilation practical and more flexible and extensible using open IRs. We further proposed to move query optimization inside the query compilation phase to enable effective cross-domain optimization. Based on these novel ideas, we designed a query compilation stack using MLIR with four new dialects for data processing. Using these dialects, we built LingoDB, a SQL-capable prototype system that uses LLVM as a code-generating backend and achieves competitive performance and low compilation times. Finally, we showed that our approach not only significantly reduces the implementation effort compared to state-of-the-art database systems but also that our design is highly extensible. For example, we can use already existing MLIR dialects, for e.g., machine-learning, to integrate and cross-optimize inference models within SQL queries.

REFERENCES

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. 2016. Tensorflow: A system for large-scale machine learning. In *12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16)*. 265–283.

[2] Maximilian Bandle and Jana Giceva. 2021. Database technology for the masses: sub-operators as first-class entities. *Proceedings of the VLDB Endowment* 14, 11 (2021), 2483–2490.

[3] Alexander Beischl, Timo Kersten, Maximilian Bandle, Jana Giceva, and Thomas Neumann. 2021. Profiling dataflow systems on multiple abstraction levels. In *Proceedings of the Sixteenth European Conference on Computer Systems*. 474–489.

[4] Sebastian Breß, Bastian Köcher, Henning Funke, Steffen Zeuch, Tilmann Rabl, and Volker Markl. 2018. Generating custom code for efficient query execution on heterogeneous processors. *The VLDB Journal* 27, 6 (2018), 797–822.

[5] Carnegie Mellon University Database Group. 2021. NoisePage. https://noise.page. Accessed: 2021-09-10.

[6] Hanfeng Chen, Joseph Vinish D'silva, Hongji Chen, Bettina Kemme, and Laurie Hendren. 2018. HorseIR: bringing array programming languages together with database query processing. In *Proceedings of the 14th ACM SIGPLAN International Symposium on Dynamic Languages*. 37–49.

[7] Anurag Gupta, Deepak Agarwal, Derek Tan, Jakub Kulesza, Rahul Pathak, Stefano Stefani, and Vidhya Srinivasan. 2015. Amazon redshift and the case for simpler data warehouses. In *Proceedings of the 2015 ACM SIGMOD international conference on management of data*. 1917–1923.

[8] Tableau Software Inc. 2021. Tableau Hyper API. https://help.tableau.com/current/api/hyper_api/en-us/index.html. Accessed: 2021-09-26.

[9] Konstantinos Karanasos, Matteo Interlandi, Doris Xin, Fotis Psallidas, Rathijit Sen, Kwanghyun Park, Ivan Popivanov, Supun Nakandal, Subru Krishnan, Markus Weimer, et al. 2019. Extending relational query processing with ML inference. *arXiv preprint arXiv:1911.00231* (2019).

[10] Alfons Kemper and Thomas Neumann. 2011. HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots. In *2011 IEEE 27th International Conference on Data Engineering*. IEEE, 195–206.

[11] Timo Kersten, Viktor Leis, and Thomas Neumann. 2021. Tidy Tuples and Flying Start: fast compilation and fast execution of relational queries in Umbra. *The VLDB Journal* (2021), 1–23.

[12] Timo Kersten and Thomas Neumann. 2020. On Another Level: How to Debug Compiling Query Engines. In *Proceedings of the Workshop on Testing Database Systems* (Portland, Oregon) *(DBTest '20)*. Association for Computing Machinery, New York, NY, USA, Article 2, 6 pages. https://doi.org/10.1145/3395032.3395321

[13] Yannis Klonatos, Christoph Koch, Tiark Rompf, and Hassan Chafi. 2014. Building efficient query engines in a high-level language. *Proceedings of the VLDB Endowment* 7, 10 (2014), 853–864.

[14] André Kohn, Viktor Leis, and Thomas Neumann. 2018. Adaptive execution of compiled queries. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*. IEEE, 197–208.

[15] André Kohn, Viktor Leis, and Thomas Neumann. 2021. Building Advanced SQL Analytics From Low-Level Plan Operators. In *Proceedings of the 2021 International Conference on Management of Data*. 1001–1013.

[16] Chris Lattner and Vikram Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* IEEE, 75–86.

[17] 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)*. IEEE, 2–14.

[18] Viktor Leis, Peter Boncz, Alfons Kemper, and Thomas Neumann. 2014. Morsel-driven parallelism: A NUMA-aware query evaluation framework for the many-core age. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*. 743–754.

[19] Prashanth Menon, Todd C. Mowry, and Andrew Pavlo. 2017. Relaxed Operator Fusion for In-Memory Databases: Making Compilation, Vectorization, and Prefetching Work Together At Last. *Proceedings of the VLDB Endowment* 11 (September 2017), 1–13. Issue 1. https://db.cs.cmu.edu/papers/2017/p1-menon.pdf

[20] Guido Moerkotte and Thomas Neumann. 2008. Dynamic programming strikes back. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*. 539–552.

[21] Ingo Müller, Renato Marroquín, Dimitrios Koutsoukos, Mike Wawrzoniak, Sabir Akhadov, and Gustavo Alonso. 2020. The collection Virtual Machine: an abstraction for multi-frontend multi-backend data analysis. In *Proceedings of the 16th International Workshop on Data Management on New Hardware*. 1–10.

[22] Thomas Neumann. 2011. Efficiently compiling efficient query plans for modern hardware. *Proceedings of the VLDB Endowment* 4, 9 (2011), 539–550.

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

[24] Thomas Neumann and Alfons Kemper. 2015. Unnesting arbitrary queries. *Datenbanksysteme für Business, Technologie und Web (BTW 2015)* (2015).

[25] OpenMP Architecture Review Board. 2021. OpenMP Application Program Interface Version. https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5-2.pdf

[26] Shoumik Palkar, James J Thomas, Anil Shanbhag, Deepak Narayanan, Holger Pirk, Malte Schwarzkopf, Saman Amarasinghe, Matei Zaharia, and Stanford InfoLab. 2017. Weld: A common runtime for high performance data analytics. In *Conference on Innovative Data Systems Research (CIDR)*, Vol. 19.

[27] Holger Pirk, Oscar Moll, Matei Zaharia, and Sam Madden. 2016. Voodoo-a vector algebra for portable database performance on modern hardware. *Proceedings of the VLDB Endowment* 9, 14 (2016), 1707–1718.

[28] CIRCT Project. 2021. "CIRCT" / Circuit IR Compilers and Tools. https://circt.llvm.org/. Accessed: 2021-12-23.

[29] LLVM Project. 2022. Kaleidoscope: Adding JIT and Optimizer Support – LLVM Optimization Passes. https://llvm.org/docs/tutorial/MyFirstLanguageFrontend/LangImpl04.html#llvm-optimization-passes. Accessed: 2022-03-22.

[30] Torch-MLIR Project. 2022. Torch-MLIR. https://github.com/llvm/torch-mlir. Accessed: 2022-03-16.

[31] Mark Raasveldt and Hannes Mühleisen. 2019. Duckdb: an embeddable analytical database. In *Proceedings of the 2019 International Conference on Management of Data*. 1981–1984.

[32] Amir Shaikhha, Yannis Klonatos, Lionel Parreaux, Lewis Brown, Mohammad Dashti, and Christoph Koch. 2016. How to architect a query compiler. In *Proceedings of the 2016 International Conference on Management of Data*. 1907–1922.

[33] SingleStore, Inc. 2021. SingleStore: The Single Database for All Data-Intensive Applications. https://singlestore.com. Accessed: 2021-09-10.

[34] Ruby Y Tahboub, Grégory M Essertel, and Tiark Rompf. 2018. How to architect a query compiler, revisited. In *Proceedings of the 2018 International Conference on Management of Data*. 307–322.

[35] The Apache Software Foundation. 2016. ClickHouse - fast open-source OLAP DBMS. https://clickhouse.tech. Accessed: 2021-09-10.