

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

Jonas Ladner¹ Lukas Döllerer² Alexis Engelke³ Michael Jungmair¹

¹ Technical University of Munich

² CedarDB

³ Univ. Grenoble Alpes, Inria, CNRS, Grenoble INP, LIG

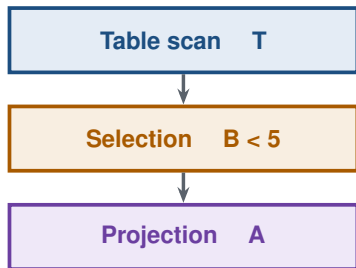
ADMS@VLDB 2026

Query compilation

Query compilation

```
select A from T where B < 5
```

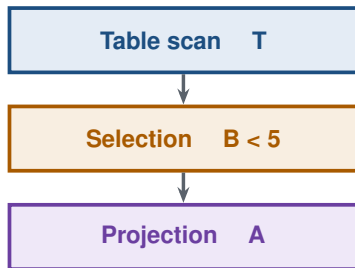
Relational operator tree



Query compilation

```
select A from T where B < 5
```

Relational operator tree



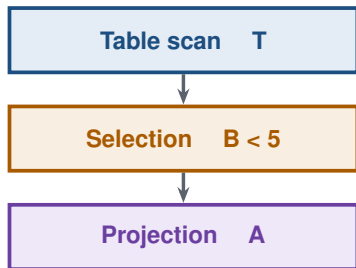
Generated code

```
for t in T:
    if t.B < 5:
        result.append([A])
```

Query compilation

```
select A from T where B < 5
```

Relational operator tree



Generated code

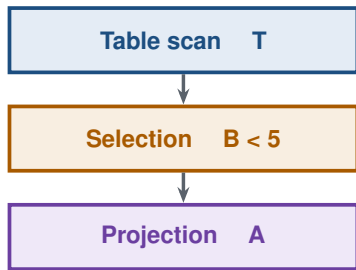
```
for t in T:  
    if t.B < 5:  
        result.append([A])
```

✓ especially effective for complex queries and expressions

Query compilation

```
select A from T where B < 5
```

Relational operator tree



Generated code

```
for t in T:
    if t.B < 5:
        result.append([A])
```

-
- ✓ especially effective for complex queries and expressions
 - ✓ *flexible*: generate arbitrary code for use cases that are difficult with interpretation

Building a compiling engine: three goals

Building a compiling engine: three goals

1. High-quality machine code

- Compiled queries have, by definition, super hot loops
- Otherwise advantage shrinks compared to interpretation

Building a compiling engine: three goals

1. High-quality machine code

- Compiled queries have, by definition, super hot loops
- Otherwise advantage shrinks compared to interpretation

2. Low compilation latencies

- Otherwise they can dominate end-to-end time for small datasets

Building a compiling engine: three goals

1. High-quality machine code

- Compiled queries have, by definition, super hot loops
- Otherwise advantage shrinks compared to interpretation

2. Low compilation latencies

- Otherwise they can dominate end-to-end time for small datasets

3. Low engineering effort

- Compilation often seems to be difficult to database engineers
- *Portability*: not only x86, but also ARM, . . .

The design space today: pick two

The design space today: pick two

	Low latency	Code quality	Low effort & portability
Generate C / C++	≈ 1 s	✓	✓

The design space today: pick two

	Low latency	Code quality	Low effort & portability
Generate C / C++	≈ 1 s	✓	✓
Emit LLVM-IR	≈ 35 ms	✓	~

The design space today: pick two

	Low latency	Code quality	Low effort & portability
Generate C / C++	≈ 1 s	✓	✓
Emit LLVM-IR	≈ 35 ms	✓	~
Use specialized backend (e.g. MIR)	$\approx 5-10$ ms	~	✗

The design space today: pick two

	Low latency	Code quality	Low effort & portability
Generate C / C++	≈ 1 s	✓	✓
Emit LLVM-IR	≈ 35 ms	✓	~
Use specialized backend (e.g. MIR)	$\approx 5-10$ ms	~	✗
Custom backend (HyPer, Umbra)	≈ 1 ms	✓	✗

The design space today: pick two

	Low latency	Code quality	Low effort & portability
Generate C / C++	≈ 1 s	✓	✓
Emit LLVM-IR	≈ 35 ms	✓	~
Use specialized backend (e.g. MIR)	$\approx 5-10$ ms	~	✗
Custom backend (HyPer, Umbra)	≈ 1 ms	✓	✗

... but also other domains like ML face similar engineering challenges

MLIR: one IR, many levels of abstraction



MLIR

MLIR: one IR, many levels of abstraction



MLIR

Meta compiler framework
supporting hierarchies of
Intermediate Representations

MLIR: one IR, many levels of abstraction



MLIR

Meta compiler framework
supporting hierarchies of
Intermediate Representations

Allows anyone to define their *own*
IR dialect

MLIR: one IR, many levels of abstraction



MLIR

Meta compiler framework
supporting hierarchies of
Intermediate Representations

Allows anyone to define their *own*
IR dialect

- Type system and operations

MLIR: one IR, many levels of abstraction



MLIR

Meta compiler framework
supporting hierarchies of
Intermediate Representations

Allows anyone to define their *own*
IR dialect

- Type system and operations
- Compiler passes

MLIR: one IR, many levels of abstraction



MLIR

Meta compiler framework
supporting hierarchies of
Intermediate Representations

Allows anyone to define their *own*
IR dialect

- Type system and operations
- Compiler passes
- Lowering to other
dialects/IRs (e.g. LLVM)

MLIR: one IR, many levels of abstraction



MLIR

Meta compiler framework
supporting hierarchies of
Intermediate Representations

Allows anyone to define their *own*
IR dialect

- Type system and operations
- Compiler passes
- Lowering to other dialects/IRs (e.g. LLVM)

```
%0 = relop.basetable "lineitem"  
%1 = relop.selection %0 (%t) {  
  %p = relop.getcol %t @l_extendedprice  
  %r = db.cmp lt, %p, %c  
}
```

(a) high-level relational operators

MLIR: one IR, many levels of abstraction



MLIR

Meta compiler framework
supporting hierarchies of
Intermediate Representations

Allows anyone to define their *own*
IR dialect

- Type system and operations
- Compiler passes
- Lowering to other dialects/IRs (e.g. LLVM)

```
%0 = relop.basetable "lineitem"  
%1 = relop.selection %0 (%t) {  
  %p = relop.getcol %t @l_extendedprice  
  %r = db.cmp lt, %p, %c  
}
```

(a) high-level relational operators

```
scf.for %i = %c0 to %n step %c1 {  
  %p = ptr.load %col[%i] : db.decimal<10,2>  
  %lt = db.cmp lt, %p, %c  
  scf.if %lt { ... }  
}
```

(b) structured control flow, database types

MLIR: one IR, many levels of abstraction



MLIR

Meta compiler framework
supporting hierarchies of
Intermediate Representations

Allows anyone to define their *own*
IR dialect

- Type system and operations
- Compiler passes
- Lowering to other dialects/IRs (e.g. LLVM)

```
%0 = relop.basetable "lineitem"  
%1 = relop.selection %0 (%t) {  
  %p = relop.getcol %t @l_extendedprice  
  %r = db.cmp lt, %p, %c  
}
```

(a) high-level relational operators

```
scf.for %i = %c0 to %n step %c1 {  
  %p = ptr.load %col[%i] : db.decimal<10,2>  
  %lt = db.cmp lt, %p, %c  
  scf.if %lt { ... }  
}
```

(b) structured control flow, database types

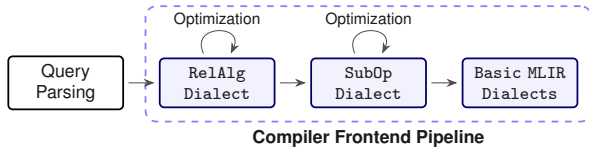
```
^loop(%i: index):  
  %q = ptr.load %col[%i] : i128  
  %lt = arith.cmpi slt, %q, %c24  
  cf.cond_br %lt, ^match, ^next
```

(c) basic blocks: cf + arith + ptr

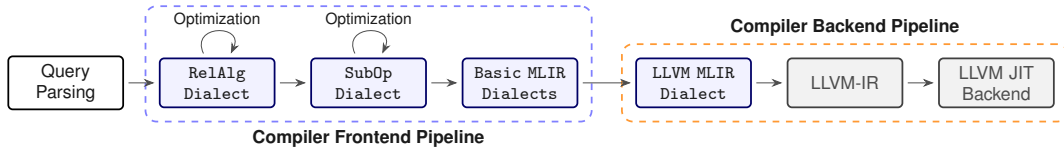
LingoDB: query optimisation and compilation, all in MLIR

Query
Parsing

LingoDB: query optimisation and compilation, all in MLIR

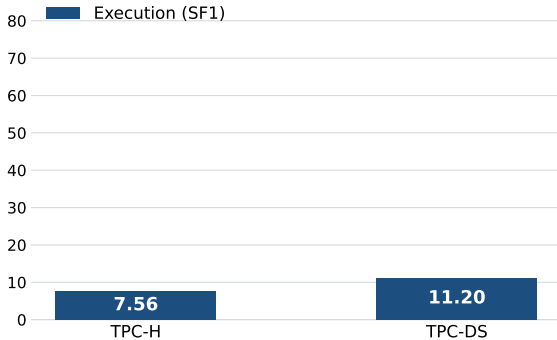


LingoDB: query optimisation and compilation, all in MLIR



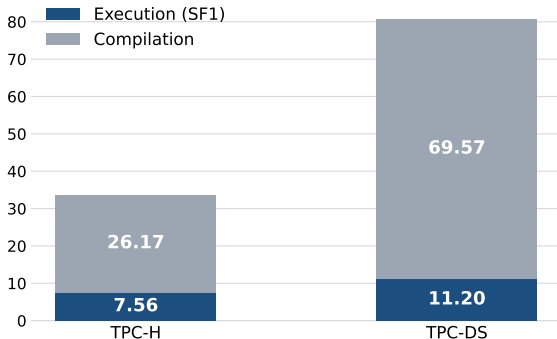
LingoDB Performance

LingoDB / LLVM: avg. latency per query (ms), x86



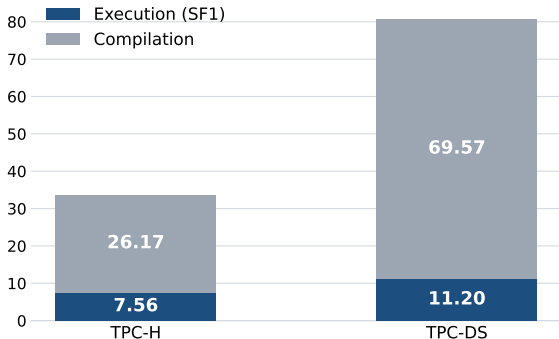
LingoDB Performance

LingoDB / LLVM: avg. latency per query (ms), x86



LingoDB Performance

LingoDB / LLVM: avg. latency per query (ms), x86

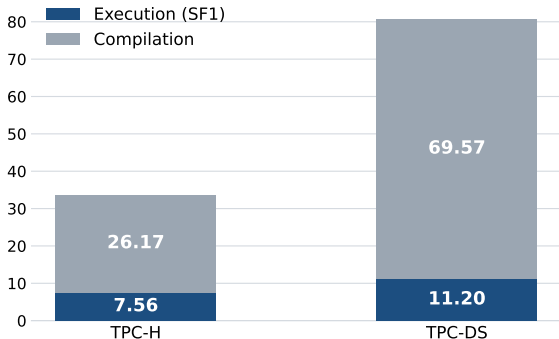


3.5× – **6.2×**

more time spent compiling than executing,
at scale factor 1

LingoDB Performance

LingoDB / LLVM: avg. latency per query (ms), x86



3.5× – 6.2×

more time spent compiling than executing,
at scale factor 1

Two independent causes

1. LLVM is a general-purpose backend.
2. MLIR's flexibility has a price.

MLIR: almost all three

	Low latency	Code quality	Low effort & portability
Generate C / C++	≈ 1 s	✓	✓
Emit LLVM-IR	≈ 35 ms	✓	~
Use specialized backend (e.g. MIR)	$\approx 5-10$ ms	~	✗
Custom backend (HyPer, Umbra)	≈ 1 ms	✓	✗
MLIR (LingoDB, Daphne, NebulaStream)	?	✓	✓

MLIR: almost all three

	Low latency	Code quality	Low effort & portability
Generate C / C++	≈ 1 s	✓	✓
Emit LLVM-IR	≈ 35 ms	✓	~
Use specialized backend (e.g. MIR)	$\approx 5-10$ ms	~	✗
Custom backend (HyPer, Umbra)	≈ 1 ms	✓	✗
MLIR (LingoDB, Daphne, NebulaStream)	≈ 50 ms	✓	✓

MLIR: almost all three

	Low latency	Code quality	Low effort & portability
Generate C / C++	≈ 1 s	✓	✓
Emit LLVM-IR	≈ 35 ms	✓	~
Use specialized backend (e.g. MIR)	$\approx 5-10$ ms	~	✗
Custom backend (HyPer, Umbra)	≈ 1 ms	✓	✗
MLIR (LingoDB, Daphne, NebulaStream)	≈ 50 ms	✓	✓

Low latency: ≈ 50 ms $\longrightarrow$ ≈ 1 ms ?

Faster backend

TPDE: a framework for building fast backends



TPDE: A Fast Adaptable Compiler Back-End Framework

Tobias Schwarz
Technical University of Munich
Munich, Germany
tobias.schwarz@tum.de

Tobias Kamm
Technical University of Munich
Munich, Germany
tobias.kamm@tum.de

Alexis Engelke
Technical University of Munich
Munich, Germany
engelke@tum.de

Faster backend

TPDE: a framework for building fast backends

Your own IR

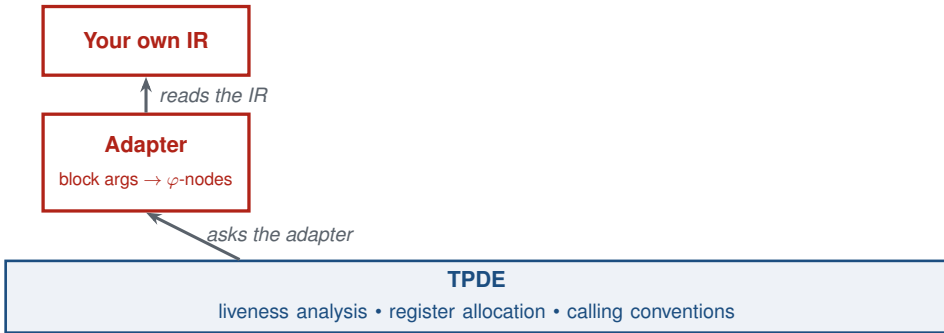
TPDE

liveness analysis • register allocation • calling conventions

Provided by: ☒ the IR author ☐ TPDE

Faster backend

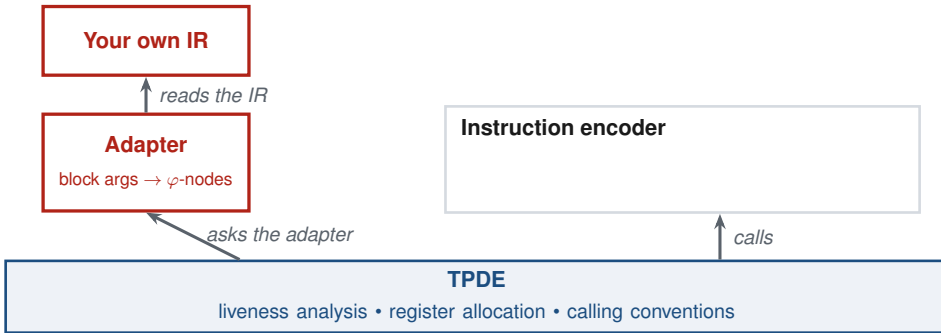
TPDE: a framework for building fast backends



Provided by: ☒ the IR author ☐ TPDE

Faster backend

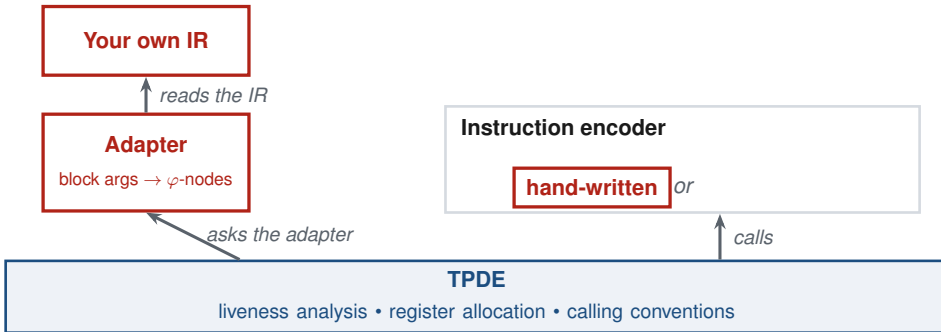
TPDE: a framework for building fast backends



Provided by: ☒ the IR author ☐ TPDE

Faster backend

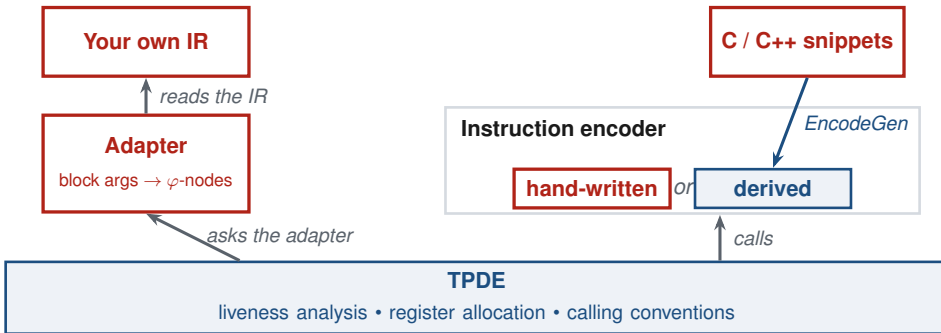
TPDE: a framework for building fast backends



Provided by: ☒ the IR author ☐ TPDE

Faster backend

TPDE: a framework for building fast backends



Provided by: ☒ the IR author ☐ TPDE

Snippets: instruction selection, for free

Snippets: instruction selection, for free

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

Snippets: instruction selection, for free

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

generate encoder from snippet

```
bool EncodeCompiler<Adaptor, Derived, BaseType, Config>::encode_arith_add_i32
    (GenericValuePart &&param_0, GenericValuePart &&param_1, ValuePart &result_0) {
    ScratchReg scratch_ax{derived()};
    ScratchReg scratch_si{derived()};
    ScratchReg scratch_di{derived()};

    do {
        auto cond0 : optional<FeMem> = encodeable_with(param_0, FE_MEM(FE_NOREG, 1, FE_AX, 0), true);

        AsnReg op0 = derived()->gval_as_reg_reuse(param_1, scratch_ax);
        // def ax has not been allocated yet
        scratch_ax.alloc(RegBank(0));
        ASMD(LEA32rn, scratch_ax.cur_reg(), FE_MEM((*cond0).base, (*cond0).scale, op3, (*cond0).off));
        break;
    }
    auto cond1 : optional<FeMem> = encodeable_with(param_1, FE_MEM(FE_AX, 0, FE_NOREG, 0), true);
    if (cond1) {
        AsnReg op1 = derived()->gval_as_reg_reuse(param_0, scratch_ax);
        // def ax has not been allocated yet
        scratch_ax.alloc(RegBank(0));
        ASMD(LEA32rn, scratch_ax.cur_reg(), FE_MEM(op1, (*cond1).scale, (*cond1).idx, (*cond1).off));
        break;
    }
    if (1) {
        AsnReg op1 = derived()->gval_as_reg_reuse(param_0, scratch_ax);
        AsnReg op3 = derived()->gval_as_reg_reuse(param_1, scratch_ax);
        // def ax has not been allocated yet
        scratch_ax.alloc(RegBank(0));
        ASMD(LEA32rn, scratch_ax.cur_reg(), FE_MEM(op1, 1, op3, 0));
        break;
    }
    } while (false);
    param_0.reset();
    param_1.reset();

    result_0.set_value(derived(), std::move(scratch_ax));
    return true;
}
```

Snippets: instruction selection, for free

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

generate encoder from snippet

```
bool EncodeCompiler<Adaptor, Derived, BaseType, Config>::encode_arith_add_i32
[GenericValuePart &&param_0, GenericValuePart &&param_1, ValuePart &result_0] {
    ScratchReg scratch_ax{derived()};
    ScratchReg scratch_si{derived()};
    ScratchReg scratch_di{derived()};

    do {
        auto cond0 : optional<FeMem> = encodeable_with(param_0, FE_MEM(FE_NOREG, 1, FE_AX, 0), true);

        AsnReg op0 = derived()->gval_as_reg_reuse(param_1, scratch_ax);
        // def ax has not been allocated yet
        scratch_ax.alloc(RegBank(0));
        ASMD(LEA32rn, scratch_ax.cur_reg(), FE_MEM((*cond0).base, (*cond0).scale, op3, (*cond0).off));
        break;
    }
    auto cond1 : optional<FeMem> = encodeable_with(param_1, FE_MEM(FE_AX, 0, FE_NOREG, 0), true);
    if (cond1) {
        AsnReg op1 = derived()->gval_as_reg_reuse(param_0, scratch_ax);
        // def ax has not been allocated yet
        scratch_ax.alloc(RegBank(0));
        ASMD(LEA32rn, scratch_ax.cur_reg(), FE_MEM(op1, (*cond1).scale, (*cond1).idx, (*cond1).off));
        break;
    }
    if (1) {
        AsnReg op1 = derived()->gval_as_reg_reuse(param_0, scratch_ax);
        AsnReg op3 = derived()->gval_as_reg_reuse(param_1, scratch_ax);
        // def ax has not been allocated yet
        scratch_ax.alloc(RegBank(0));
        ASMD(LEA32rn, scratch_ax.cur_reg(), FE_MEM(op1, 1, op3, 0));
        break;
    }
    } while (false);
    param_0.reset();
    param_1.reset();

    result_0.set_value(derived(), std::move(scratch_ax));
    return true;
}
```

Snippets: instruction selection, for free

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

generate encoder from snippet

```
bool compile_arith_addi(IRInstRef inst) {
    auto lhs = val_ref(inst->getOperand(0));
    auto rhs = val_ref(inst->getOperand(1));
    auto res = result_ref(inst);
    switch (bitwidth) {
        case 32: return encode_addU32(...);
        ...
    }
}
```

use generated encoder

```
bool EncodeCompiler<Adaptor, Derived, BaseType, Config>::encode_arith_addi32
    (GenericValuePart &&param_0, GenericValuePart &&param_1, ValuePart &result_0) {
    ScratchReg scratch_ax{derived()};
    ScratchReg scratch_si{derived()};
    ScratchReg scratch_di{derived()};

    do {
        auto cond0 : optional<FeMem> = encodeable_with(param_0, FE_MEM(FE_NOREG, 1, FE_AX, 0), true);
        AsnReg op0 = derived()->gval_as_reg_reuse(param_1, scratch_ax);
        // def ax has not been allocated yet
        scratch_ax.alloc(RegBank(0));
        ASMD(LEA32rn, scratch_ax.cur_reg(), FE_MEM((*cond0).base, (*cond0).scale, op3, (*cond0).off));
        break;
    }
    auto cond1 : optional<FeMem> = encodeable_with(param_1, FE_MEM(FE_AX, 0, FE_NOREG, 0), true);
    if (cond1) {
        AsnReg op1 = derived()->gval_as_reg_reuse(param_0, scratch_ax);
        // def ax has not been allocated yet
        scratch_ax.alloc(RegBank(0));
        ASMD(LEA32rn, scratch_ax.cur_reg(), FE_MEM(op1, (*cond1).scale, (*cond1).idx, (*cond1).off));
        break;
    }
    if (1) {
        AsnReg op1 = derived()->gval_as_reg_reuse(param_0, scratch_ax);
        AsnReg op3 = derived()->gval_as_reg_reuse(param_1, scratch_ax);
        // def ax has not been allocated yet
        scratch_ax.alloc(RegBank(0));
        ASMD(LEA32rn, scratch_ax.cur_reg(), FE_MEM(op1, 1, op3, 0));
        break;
    }
    } while (false);
    param_0.reset();
    param_1.reset();

    result_0.set_value(derived(), std::move(scratch_ax));
}
```

Custom database operations, without custom assembly

```
// does the 16-bit filter in the pointer's
// upper bits match this hash?
bool ptr_tag_matches(void* ptr, uint64_t hash,
                     void* bloomMaskPtr) {
    uint64_t slot = hash >> 53;
    uint16_t tag = ((uint16_t*)bloomMaskPtr)[slot];
    uint16_t entry = (uint16_t)(uintptr_t) ptr;
    return (tag & (entry ^ 0xFFFF)) == 0;
}
```

Custom database operations, without custom assembly

```
// does the 16-bit filter in the pointer's
// upper bits match this hash?
bool ptr_tag_matches(void* ptr, uint64_t hash,
                     void* bloomMaskPtr) {
    uint64_t slot = hash >> 53;
    uint16_t tag = ((uint16_t*)bloomMaskPtr)[slot];
    uint16_t entry = (uint16_t)(uintptr_t) ptr;
    return (tag & (entry ^ 0xFFFF)) == 0;
}
```

- one C function for defining the semantics
- *portable*: machine-code on every ISA

Custom database operations, without custom assembly

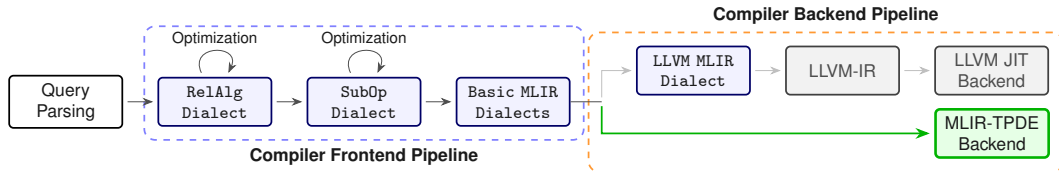
```
// does the 16-bit filter in the pointer's
// upper bits match this hash?
bool ptr_tag_matches(void* ptr, uint64_t hash,
                    void* bloomMaskPtr) {
    uint64_t slot = hash >> 53;
    uint16_t tag = ((uint16_t*)bloomMaskPtr)[slot];
    uint16_t entry = (uint16_t)(uintptr_t) ptr;
    return (tag & (entry ^ 0xFFFF)) == 0;
}
```

- one C function for defining the semantics
- *portable*: machine-code on every ISA

What still has to be written per architecture

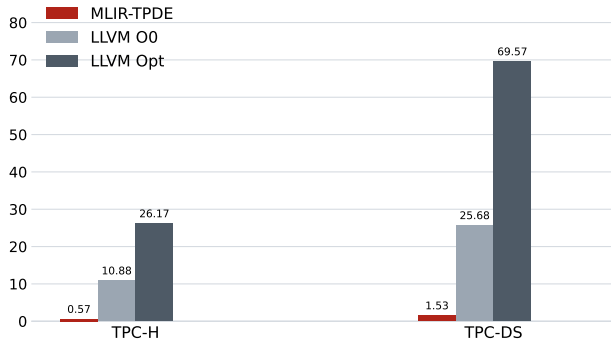
- Calling conventions
- Control flow
- Symbol resolution
- Instruction fusion

Integration into LingoDB



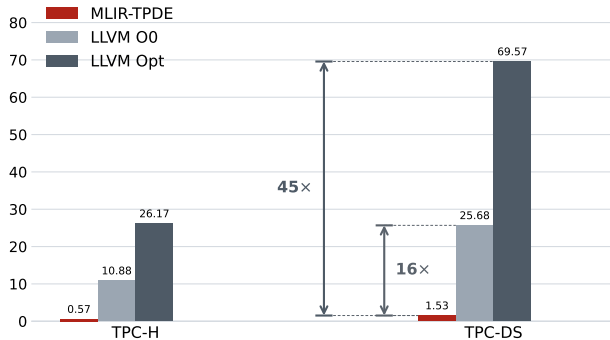
The backend: compilation latency

Total backend latency per query (ms), x86 -- lower is better

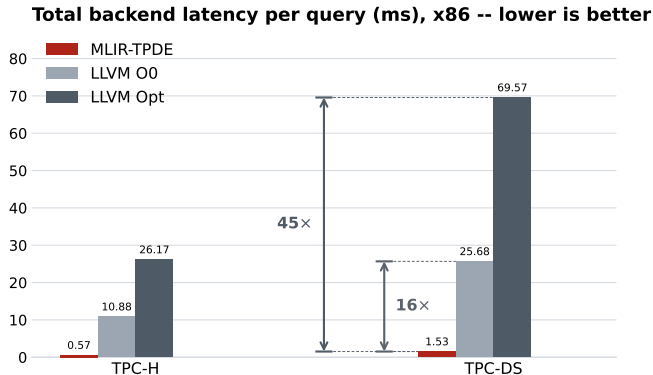


The backend: compilation latency

Total backend latency per query (ms), x86 -- lower is better



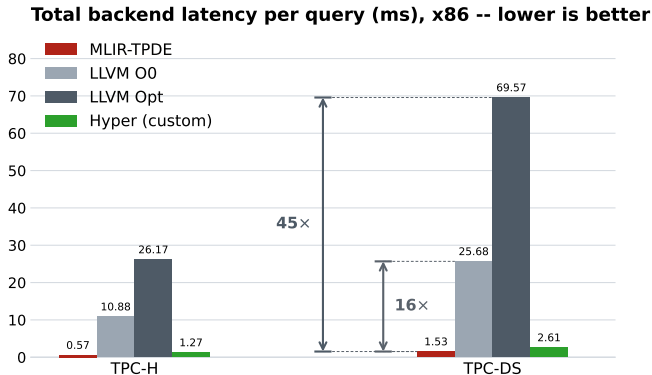
The backend: compilation latency



Why?

- TPDE: faster code generation than LLVM
- much less preparation time in MLIR (13 – 16 $\times$)

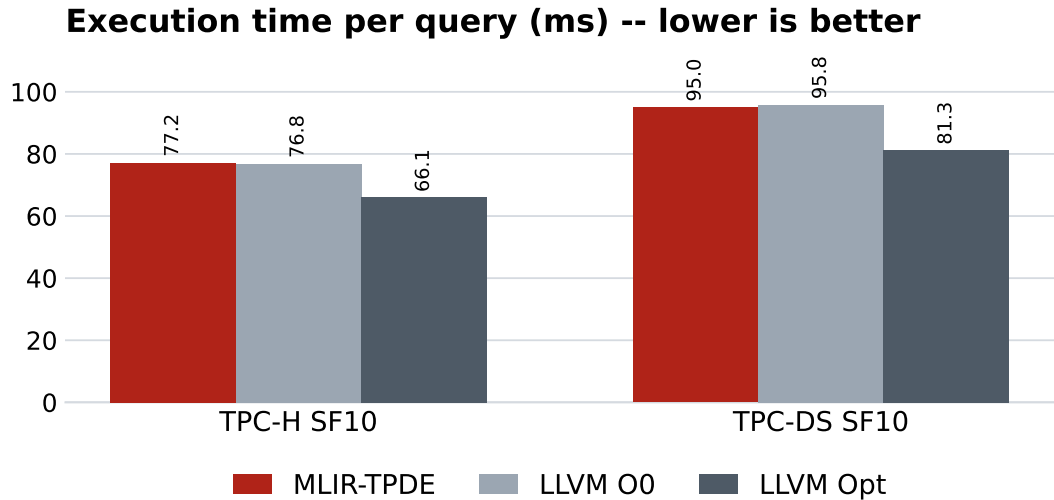
The backend: compilation latency



Why?

- TPDE: faster code generation than LLVM
- much less preparation time in MLIR (13 – 16 $\times$)
- ✓ even beats Hyper's hand-written custom backend

...and the code it generates is good



Getting MLIR out of the critical path

Getting MLIR out of the critical path

1. Context pooling (-0.5ms)

Getting MLIR out of the critical path

1. Context pooling (-0.5ms)

2. No verification after every pass

Getting MLIR out of the critical path

1. Context pooling (-0.5ms)

2. No verification after every pass

3. Cheaper pattern rewrite logic

Getting MLIR out of the critical path

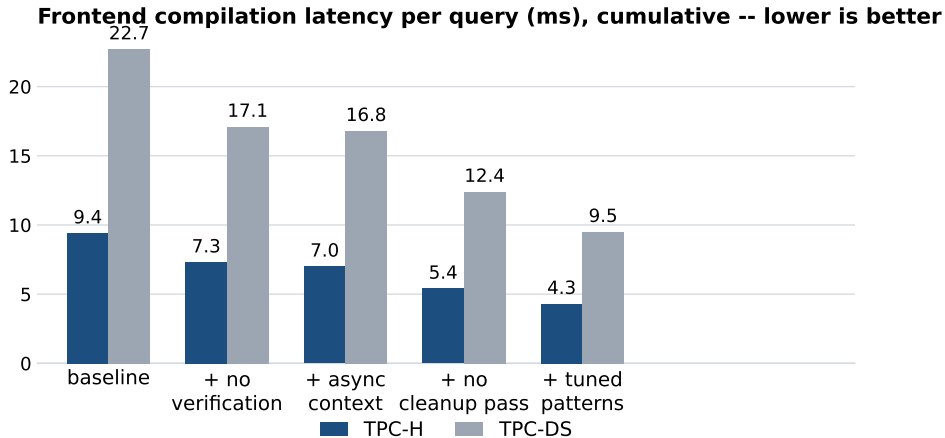
1. Context pooling (-0.5ms)

2. No verification after every pass

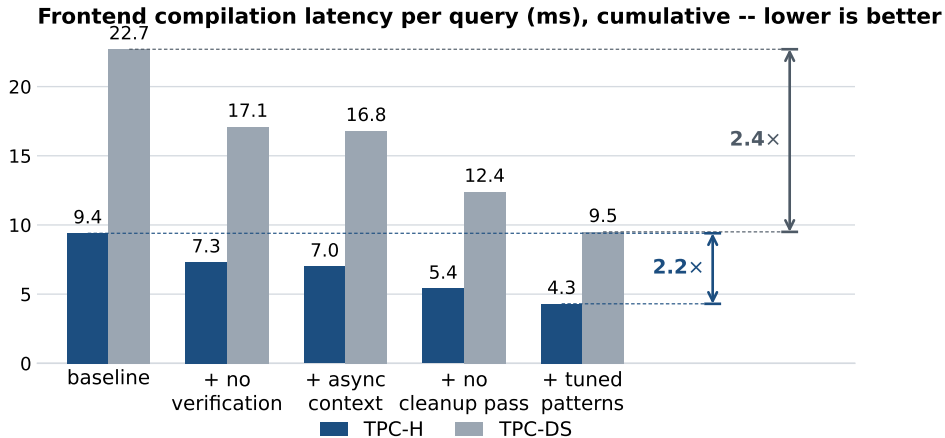
3. Cheaper pattern rewrite logic

4. Proactive folding instead of cleanup

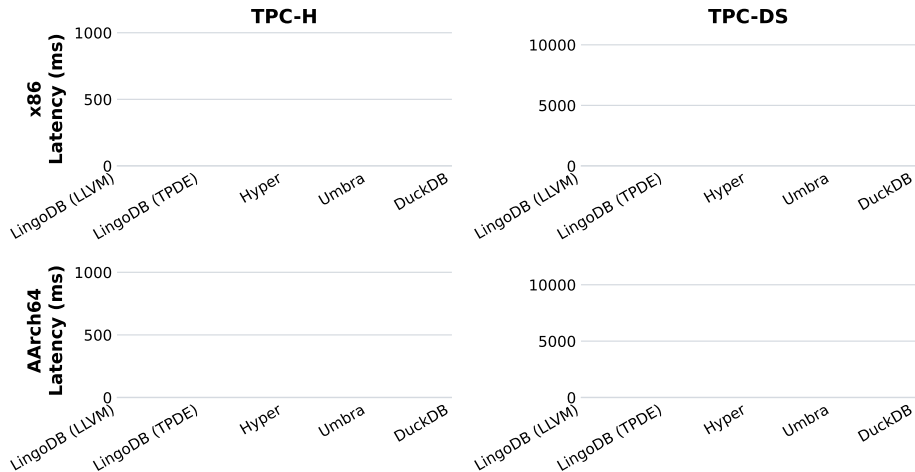
Halving MLIR's intrinsic overhead



Halving MLIR's intrinsic overhead

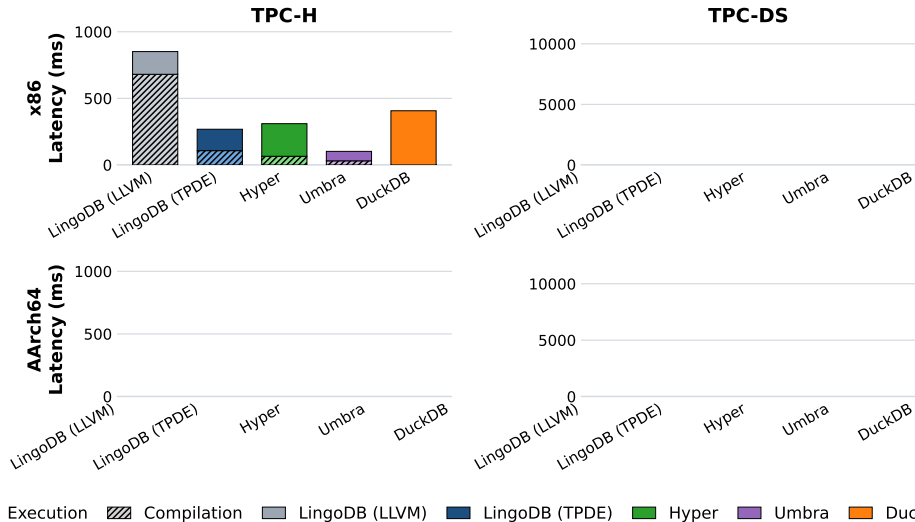


End-to-end latency at scale factor 1

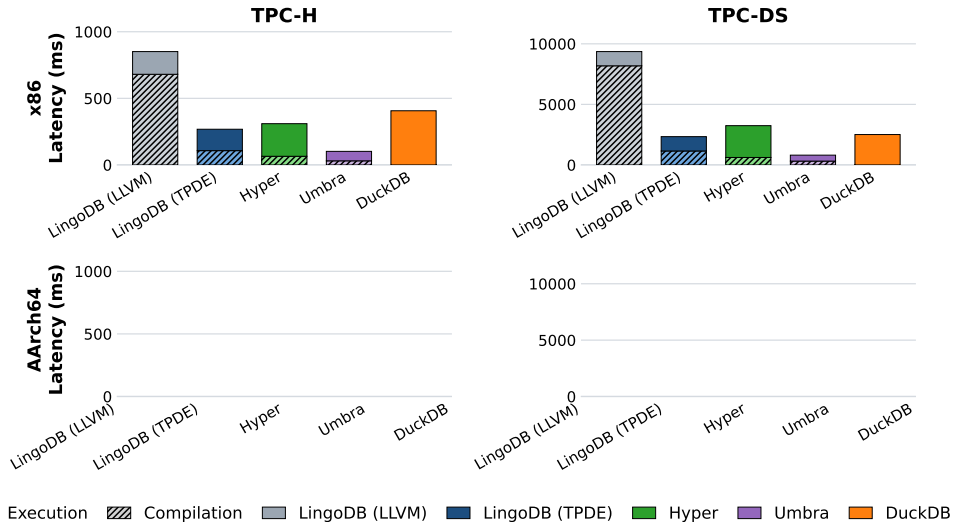


Execution Compilation LingoDB (LLVM) LingoDB (TPDE) Hyper Umbra DuckDB

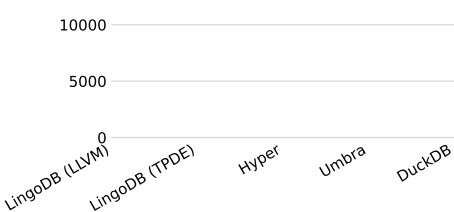
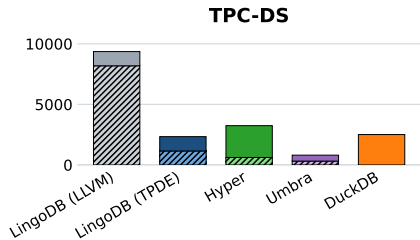
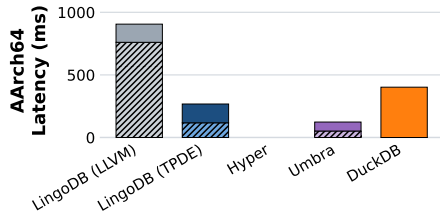
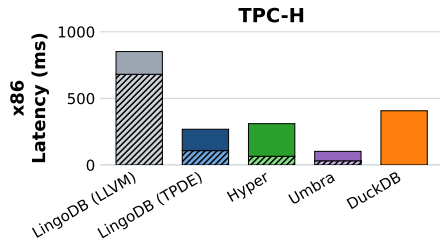
End-to-end latency at scale factor 1



End-to-end latency at scale factor 1

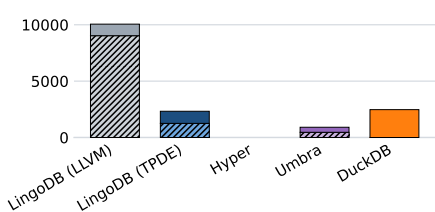
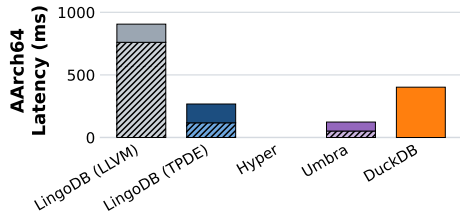
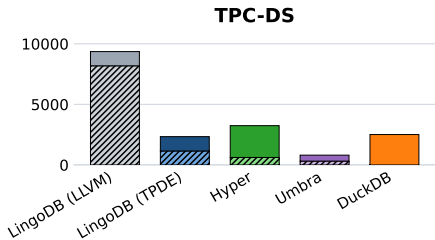
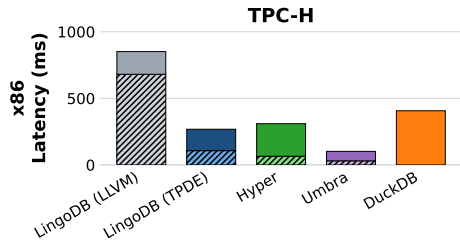


End-to-end latency at scale factor 1



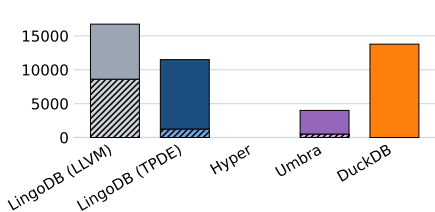
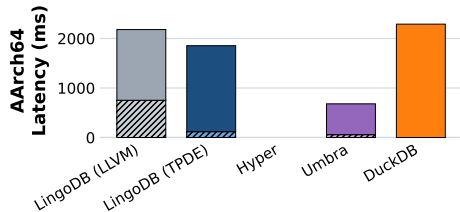
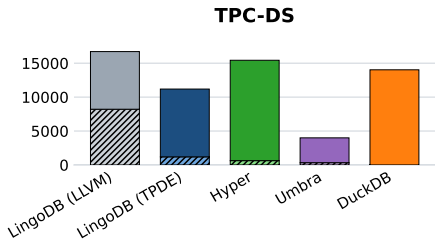
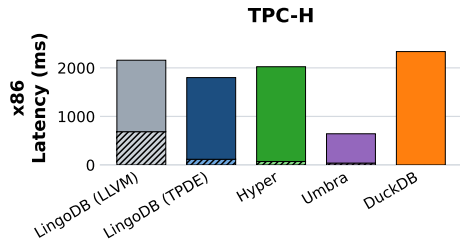
Execution Compilation LingoDB (LLVM) LingoDB (TPDE) Hyper Umbra DuckDB

End-to-end latency at scale factor 1



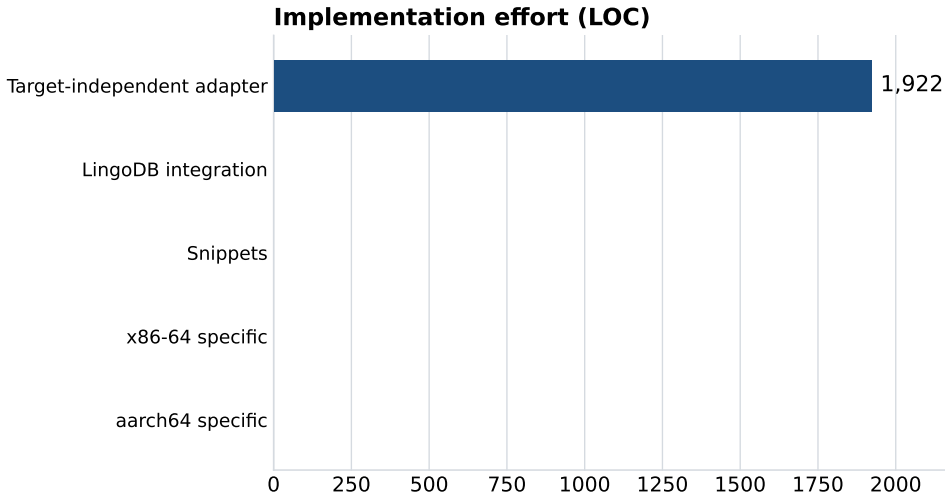
Execution Compilation LingoDB (LLVM) LingoDB (TPDE) Hyper Umbra DuckDB

End-to-end latency at scale factor 10

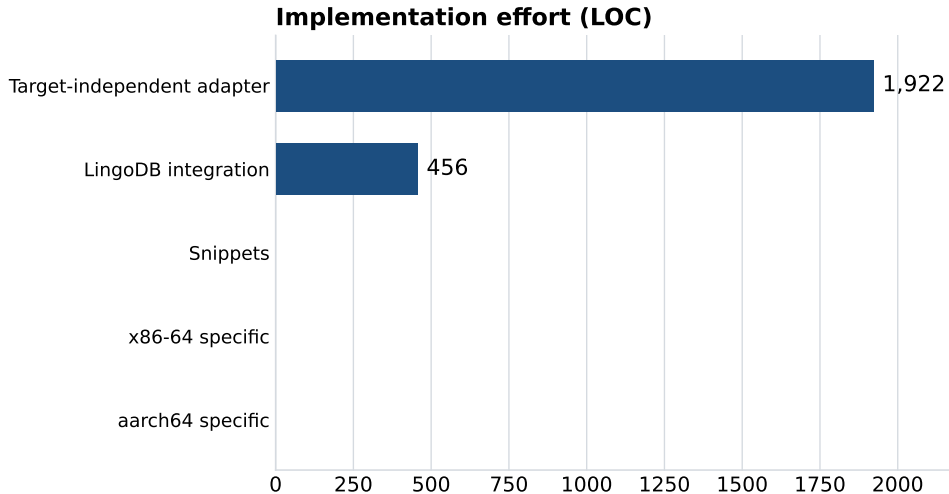


Execution Compilation LingoDB (LLVM) LingoDB (TPDE) Hyper Umbra DuckDB

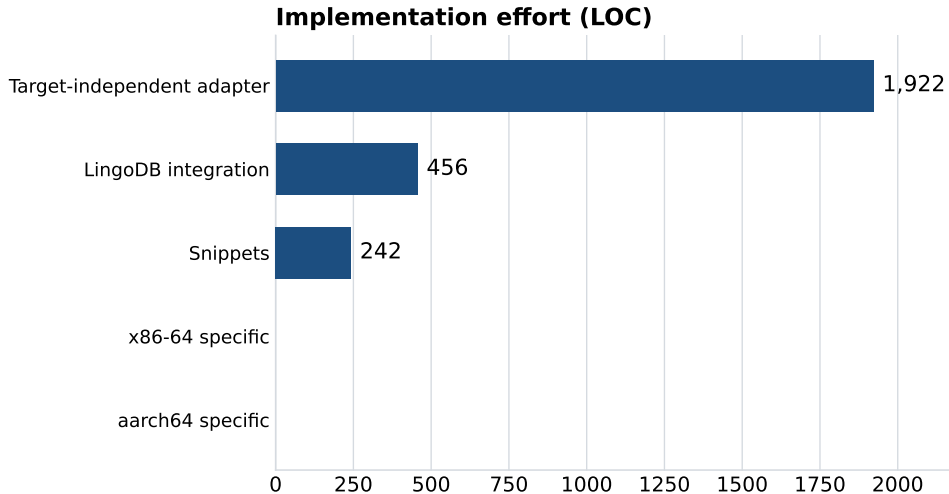
Implementation effort



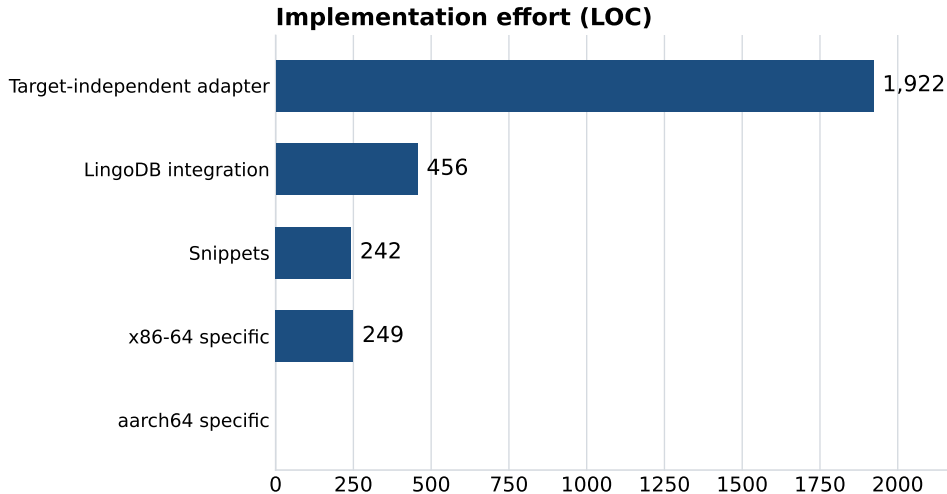
Implementation effort



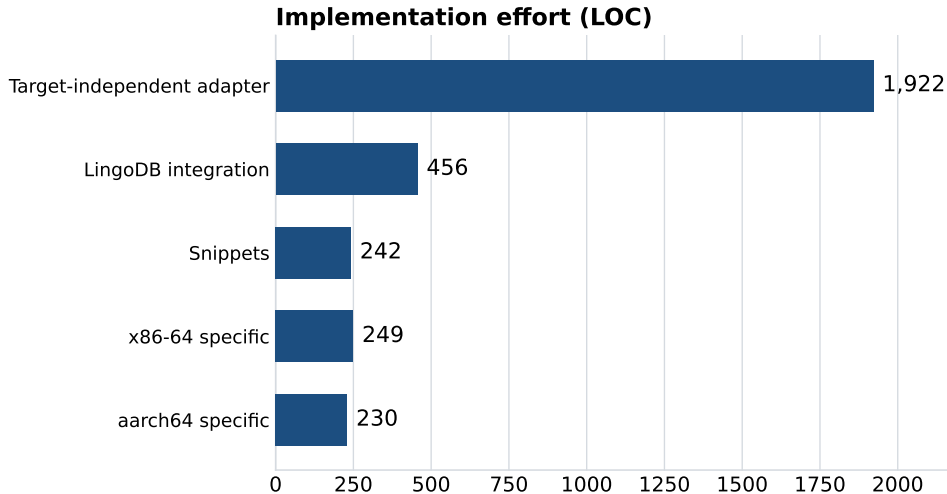
Implementation effort



Implementation effort



Implementation effort



Conclusion

Conclusion

You can have all three with MLIR & TPDE

- Low enough compilation latency
- High-quality code
- Low engineering effort and portability

Conclusion

You can have all three with MLIR & TPDE

- Low enough compilation latency
- High-quality code
- Low engineering effort and portability

Reusable

- Everything open-sourced through LingoDB
- Other MLIR users can also benefit

Conclusion

You can have all three with MLIR & TPDE

- Low enough compilation latency
- High-quality code
- Low engineering effort and portability

Reusable

- Everything open-sourced through LingoDB
- Other MLIR users can also benefit

Future

- LingoDB already runs fast on GPUs (stay tuned)
- compilation times (LLVM/NVRTC) can again dominate
- MLIR-TPDE backend for Nvidia/AMD/...

→ Talk to me if you are interested

Conclusion

You can have all three with MLIR & TPDE

- Low enough compilation latency
- High-quality code
- Low engineering effort and portability

Reusable

- Everything open-sourced through LingoDB
- Other MLIR users can also benefit

Future

- LingoDB already runs fast on GPUs (stay tuned)
- compilation times (LLVM/NVRTC) can again dominate
- MLIR-TPDE backend for Nvidia/AMD/...

→ Talk to me if you are interested

Try it out!

Code: github.com/lingo-db/lingo-db

Website: www.lingo-db.com

The screenshot shows the LingoDB web interface. At the top, there's a LingoDB logo and a note: "This webinterface is for demo purposes only, and especially not suited for benchmarking." Below this, a SQL query is entered in a text area. The query is a TPC-H Query 3, which selects orderkey, revenue (calculated as sum of extendedprice minus discount), orderdate, and shippriority from the customer and orders tables. The interface includes buttons for "Execute Query", "Options", and a user identifier "LingoDB@5c3095a". Below the query area, there are tabs for "QueryPlan", "SubOperators", "MLIR (RelAlg)", "MLIR (SubOp)", "MLIR (HL Imperative)", and "MLIR (LL Imperative)". The "Execution" tab is active, showing performance metrics: "Execution: 41.2 ms", "Optimization: 1.2 ms", and "Compilation: 4.6 ms". At the bottom, a table displays the query results with columns: L_orderkey, revenue, o_orderdate, and o_shippriority. The table contains four rows of data.

```
1  -- TPC-H Query 3
2
3  select
4      l_orderkey,
5      sum(l_extendedprice * (1 - l_discount)) as revenue,
6      o_orderdate,
7      o_shippriority
8  from
9      customer,
10     orders,
```

▶ Execute Query TPC-H (SF1) ▼ Select Query ▼ Options ▼ LingoDB@5c3095a

Result: QueryPlan SubOperators MLIR (RelAlg) MLIR (SubOp) MLIR (HL Imperative) MLIR (LL Imperative)

Execution: 41.2 ms Optimization: 1.2 ms Compilation: 4.6 ms

L_orderkey	revenue	o_orderdate	o_shippriority
2456423	406181.0111	1995-03-05	0
3459808	405838.6989	1995-03-04	0
492164	390324.0610	1995-02-19	0
1188320	384537.9359	1995-03-09	0

Thank you!

Questions?

Code: `github.com/lingo-db/lingo-db`

Website: `www.lingo-db.com`