Tenth LLVM-CGO Workshop 2026

Polymer: An Explainable Database Execution Engine Based on MLIR

A Compiler-Centric Approach to Transparent and Extensible Database Systems

Yizhe Zhang, Bocheng Han, Zhengyi Yang

January, 31

Why we use MLIR
to create a
Database Execution Engine?

Some awesome database use MLIR/LLVM

HyPer

UMBRA

noise page

LingoDB

Research Motivation

Limited Extensibility

Evaluating individual operator implementations typically requires modifying source code , making experimentation costly and time-consuming.

- Developing new Query Optimizers is difficult to validate
- New data formats require complete SQL parser integration

Limited Explainability

Database systems suffer from limited explainability , constraining database operation reuse across language boundaries.

- Traditional systems provide limited operator-level visibility
- Database operation reuse constrained by language boundaries

LLVM Ecosystem Opportunity

LLVM provides mature debugging infrastructure that can help database developers understand optimization effects .

- ✓ Comprehensive debugging tools
- ✓ Multi-level IR representation
- ✓ Performance profiling capabilities

What Database Design we implement with MLIR?

Database Execution Architecture

Multi-Stage Architecture

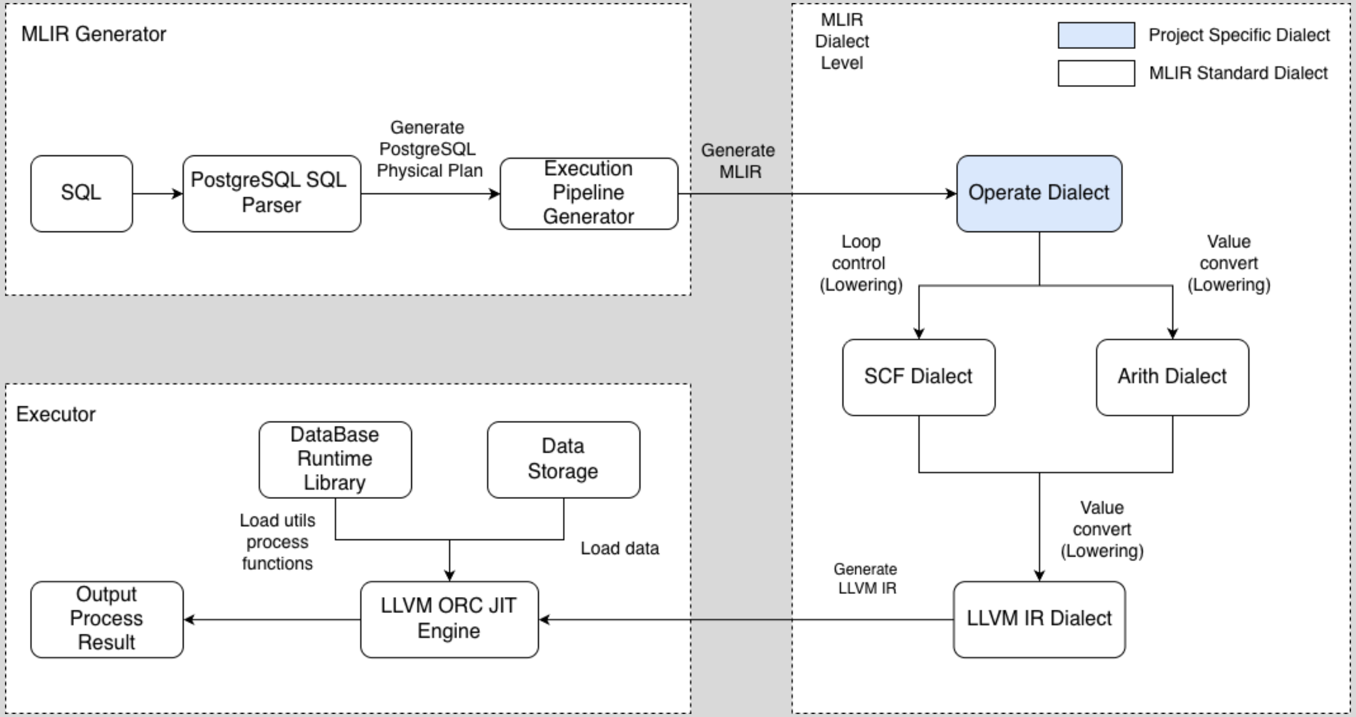
Modern database systems employ a three-stage architecture to transform SQL queries into efficient executable code:

1. **SQL Parsing & Semantic Analysis**
   Transform declarative queries into logical plans
2. **Query Optimization**
   Cost-based optimization, join ordering, operator selection
3. **Query Execution**
   Orchestrate dataflow between operators

Execution Strategies

1. **Pipeline Execution**
   Streaming data processing to reduce materialization
2. **Vectorized Processing**
   Fixed-size batches for SIMD optimization
3. **JIT Compilation**
   Convert SQL execution plan to LLVM IR

Polymer Architecture Overview



PostgreSQL Integration

Accepts physical query plans from PostgreSQL optimizer, transforming them into MLIR modules.

MLIR Representation

Database operations modeled as composable MLIR operators enabling fine-grained optimization.

LLVM JIT Execution

Lowered to LLVM IR and executed via ORC JIT runtime for high-performance execution.

Storage Formats

Pluggable executor interface supports multiple storage layouts:

Apache Arrow | Apache Parquet | TPC-H tbl(Text)

MLIR: Operate Dialect Design

Database-Specific Operations

Scan Operations

operate.scanInit

Initialize scan context for table schema

operate.scanNext

Retrieve data in batches

HashJoin Operations

operate.hashJoinInit
operate.hashJoinBuild
operate.hashJoinProbe
operate.hashJoinGetUnmatchedBuild

Aggregation

Non-Grouped: plainAggregate

Grouped: hashAggregate

Three-stage pattern: Init → Source → Sink

Selection & Projection

operate.filter

Applies predicates, produces selection vectors

Sort Operations

operate.sortInit

operate.sortSource

operate.sortSink

Materialize

Materializes intermediate results when pipeline breaking is necessary

Key Design Principle

Each operator maps to a corresponding MLIR operation, enabling **fine-grained debugging** and **systematic optimization** across operator boundaries.

Pipeline Execution Model

From Physical Plans to Push-Based Pipelines

```sql
select
        l_returnflag,
        l_linestatus,
        sum(l_quantity) as sum_qty,
        sum(l_extendedprice) as sum_base_price,
        sum(l_extendedprice * (1 - l_discount)) as sum_disc_price,
        sum(l_extendedprice * (1 - l_discount) * (1 + l_tax)) as sum_charge,
        avg(l_quantity) as avg_qty,
        avg(l_extendedprice) as avg_price,
        avg(l_discount) as avg_disc,
        count(*) as count_order
from
        lineitem
where
        l_shipdate <= date '1998-12-01' - interval '90' day
group by
        l_returnflag,
        l_linestatus
order by
        l_returnflag,
        l_linestatus
```

TPC-H Q1 Pipeline Decomposition

PostgreSQL physical plan decomposed into three pipelined functions:

pipeline_0 · Scan & Aggregation Build

Init context → Scan lineitem → Apply filter (l_shipdate ≤ '1998-12-01') → Push to aggregation state

pipeline_1 · Aggregation Finalization & Sort Build

Consume hash table → Produce aggregated results → Feed to sort operator

pipeline_2 · Sort Output

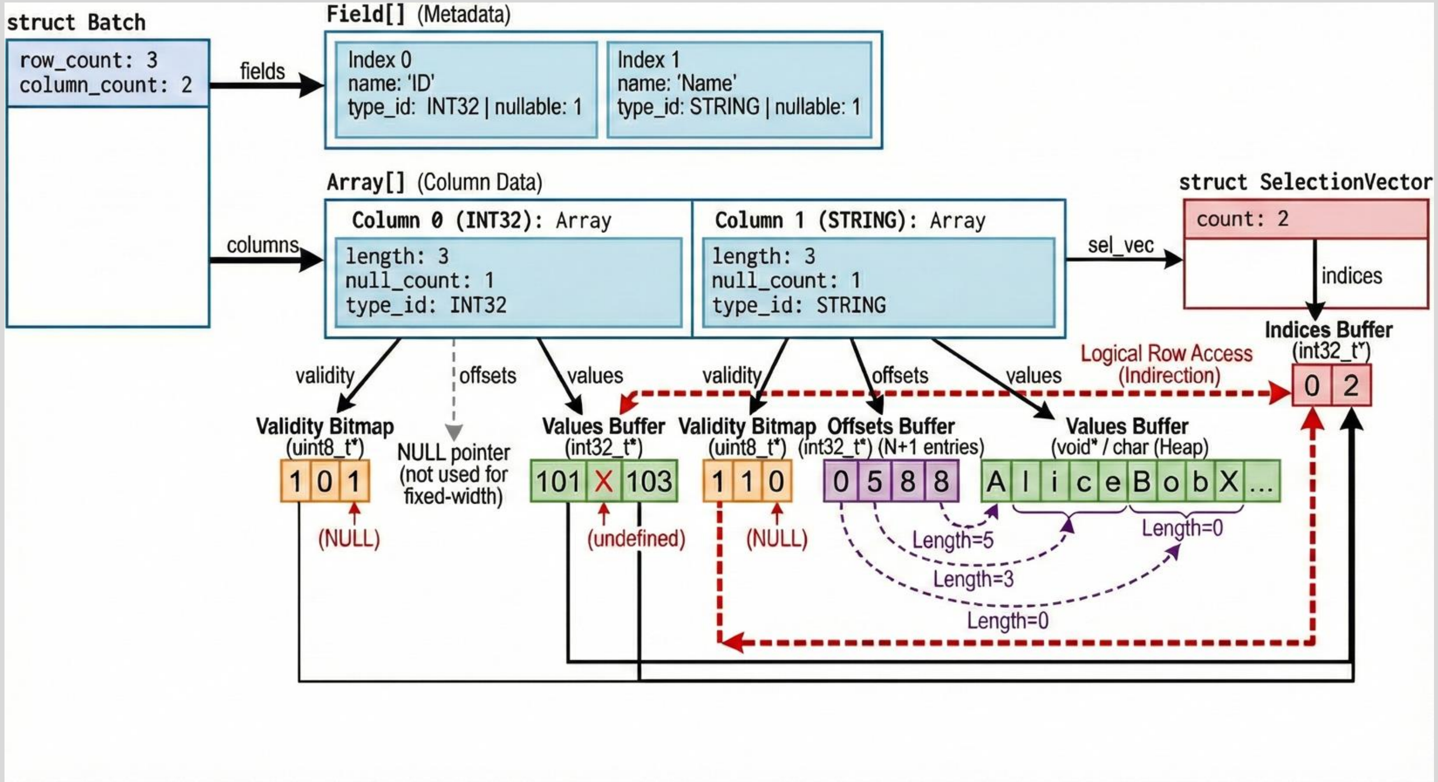
Perform sorting → Produce final ordered result batches

Context Orchestration

The main function orchestrates pipelines by passing context objects, ensuring state preservation across boundaries.

```
module {
  func.func @pipeline_0(%arg0: index) -> !operate.hashaggregatecontext {
    %0 = operate.hashAggregateInit([{column_name = "l_returnflag", varattno = 8 : i32, vartype = 1042 : i32, vartypmod = 5 : i32}
    %1 = operate.scanInit {batch_size = 2048 : i64, cols = ["l_orderkey", "l_partkey", "l_suppkey", "l_linenumber", "l_quantity",
    scf.while : () -> () {
      %2 = operate.check_hasMoreBatch(%1) : (!operate.scancontext) -> i1
      scf.condition(%2)
    } do {
      %2 = operate.scanNext(%1) : (!operate.scancontext) -> !operate.batch
      %3 = operate.filter %2 {predicate = [{col = "l_shipdate", const_i32 = -486 : i32, const_str = "'1998-09-02'", const_type =
      operate.hashAggregateSource(%3, %0, [{column_name = "l_returnflag", varattno = 8 : i32, vartype = 1042 : i32, vartypmod = 5
      scf.yield
    }
    operate.scanDestroy(%1) : (!operate.scancontext) -> ()
    return %0 : !operate.hashaggregatecontext
  }
  func.func @pipeline_1(%arg0: !operate.hashaggregatecontext) -> !operate.sortcontext {
    %0 = operate.hashAggregateSink(%arg0, [{column_name = "l_returnflag", varattno = 8 : i32, vartype = 1042 : i32, vartypmod = 5
    %1 = operate.sortInit(%0, [[1042 : i32, 1 : i32], [1042 : i32, 1 : i32]]) -> !operate.sortcontext
    operate.sortSource(%1, %0, [[1042 : i32, 1 : i32], [1042 : i32, 1 : i32]], [[0 : i32, true, false], [1 : i32, true, false]])
    return %1 : !operate.sortcontext
  }
  func.func @pipeline_2(%arg0: !operate.sortcontext) -> !operate.sortcontext {
    %0 = operate.sortSink([[1042 : i32, 1 : i32], [1042 : i32, 1 : i32]], [[true, false], [true, false]], %arg0) -> !operate.batc
    return %arg0 : !operate.sortcontext
  }
  func.func @main(%arg0: index) {
    %0 = call @pipeline_0(%arg0) : (index) -> !operate.hashaggregatecontext
    %1 = call @pipeline_1(%0) : (!operate.hashaggregatecontext) -> !operate.sortcontext
    %2 = call @pipeline_2(%1) : (!operate.sortcontext) -> !operate.sortcontext
    return
  }
}
```

Data Exchange Format



Field (Metadata Schema)

Defines column schema (name, type, nullability) to ensure type-safe data transfer between operators.

Array

Columnar storage optimized for SIMD, utilizing bitmaps, offsets, and contiguous buffers for performance.

Selection Vector

Database operations modeled as composable MLIR operators enabling fine-grained optimization.

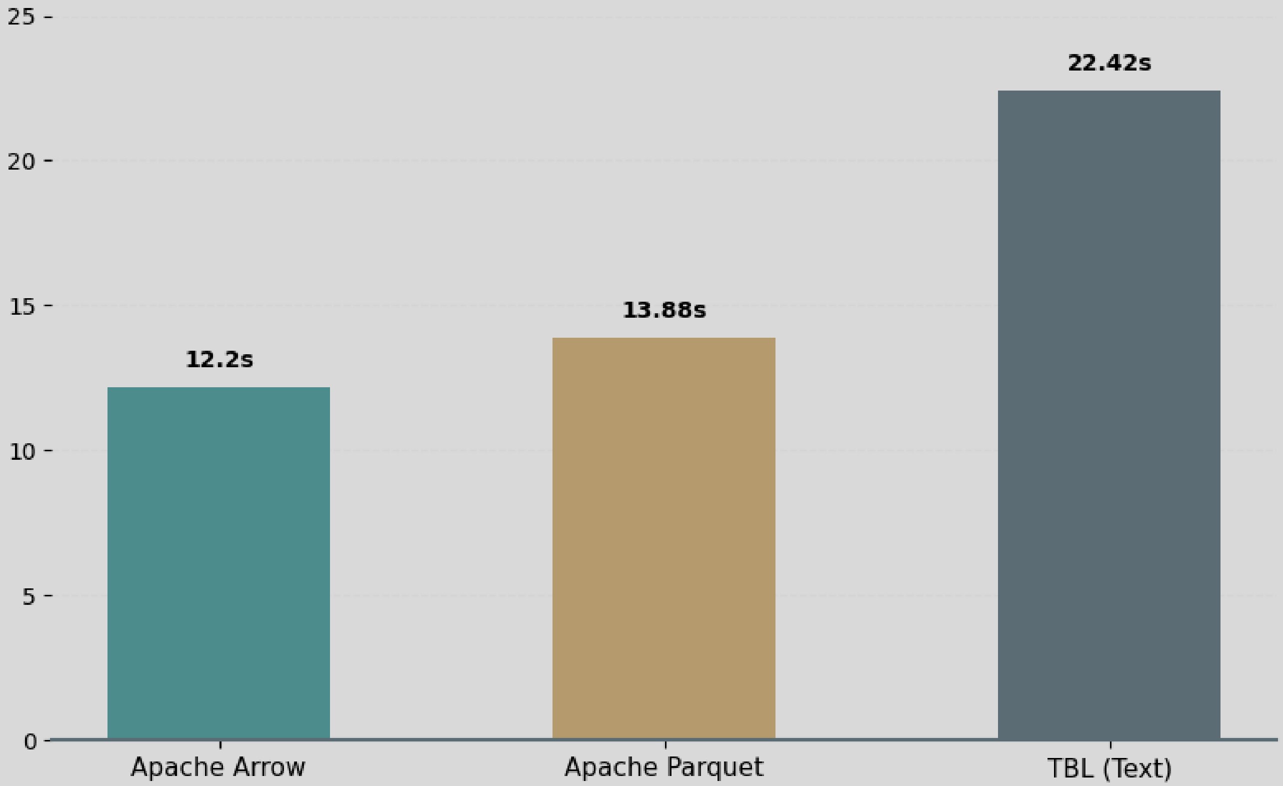
So how well does it work?

Storage Format Interface

& Performance Comparison

TPC-H Dataset Load Time Comparison

Scale Factor 1 · 8,661,245 tuples · Lower is Better



Pluggable Interface

Decouples execution from storage , enabling direct I/O performance comparison.

Implement the Executor interface for new formats.

Performance Insights

Arrow vs. TBL: 45.6% faster

Zero-copy memory mapping, no parsing overhead

Parquet vs. TBL: 38.2% faster

Columnar organization, efficient batch processing

Key Optimizations

- ✓ INT32/INT64: Batch vectorized copy
- ✓ DATE32: Specialized batch conversion
- ✓ STRING: Pre-allocation + bulk copy

Sort Operator Performance

PDQSort vs. std::sort Comparison

Execution Time Across TPC-H Queries

Log Scale · Lower is Better

Measurement Methodology

Timestamp operations injected in MLIR to isolate sort latency:

```
%start_time = operate.getCurrentTimestamp()
%end_time = operate.getCurrentTimestamp()
operate.calculateDurationMs(...)
```

Performance Results

Q13 (SF3): 1.8× faster

PDQSort: 59,739μs vs. std::sort: 109,025μs

Q10 (SF3): Substantial improvement

Q18: Maintains slight edge

Why PDQSort?

Pattern-defeating quicksort optimized for **real-world data patterns**. Validated for complex analytical queries.

Observability & Debugging

Fine-Grained Performance Analysis

MLIR-Based Observability

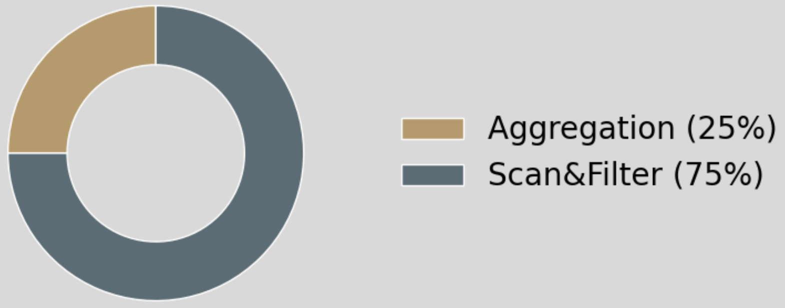
Transforms system observability by exposing each operator as a distinct compilation unit .

- ✓ Operator-level performance profiling
- ✓ MLIR representation inspection
- ✓ LLVM IR inspection
- ✓ Built-in profiling passes

Debugging Workflow

1. Examine MLIR representation
2. Apply profiling passes
3. Inspect lowered LLVM IR
4. Identify bottlenecks

TPC-H Q1 Performance Breakdown



Execution Time

~770 ms

Average of 5 runs

Processed

~2930 batches

6,001,215 tuples

Key Findings

- **Memory access** during Scan is the primary bottleneck
- **LLVM IR inspection** confirms efficient translation

Conclusion & Future Work

Key Contributions

Polymer represents a new approach leveraging MLIR's multi-level IR.

1. **Fair Algorithm Comparison**
   Unified platform targeting common operators
2. **Data Format Performance Evaluation**
   Pluggable storage interface
3. **Comprehensive Observability**
   Fine-grained overserve with MLIR tools

Future Work

Multiple Query Optimizer Adapters

Extend beyond PostgreSQL to support Apache Calcite, DuckDB, custom optimizers .

MLIR/LLVM Toolchain Integration

Explore pass pipelines, PGO, sanitizers for database workloads.

Advanced Performance Analysis

Develop automated profiling tools for query plan analysis.

Thank You！

Questions & Discussion