# LingoDB-CT: Understanding LingoDB's Inner Workings

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

## Abstract

While compiling query engines can be highly efficient, especially for complex queries, they also have the reputation of being too difficult to understand, debug, and profile. However, this is not necessarily the case if supported by the right architecture and tools.

With this demonstration, we want to show this for our own compiling query engine LingoDB. For this purpose, we built an instrumentation and visualization framework called LingoDB-CT, which aggregates data from multiple angles and visualizes it to reveal the bigger picture. Through two interactive demonstration scenarios, using publicly hosted web applications, we show that LingoDB is indeed easy to understand and profile.

## CCS Concepts

• **Information systems** → **Database query processing**.

## Keywords

Understanding, Profiling, Query Optimization, Query Compilation

**ACM Reference Format:**
Michael Jungmair. 2025. LingoDB-CT: Understanding LingoDB's Inner Workings. In *Companion of the 2025 International Conference on Management of Data (SIGMOD-Companion '25), June 22–27, 2025, Berlin, Germany*. ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3722212.3725111

## 1 Introduction

Around 15 years ago, compiling SQL queries into efficient machine code emerged as a new approach to query execution [7]. By avoiding virtual function calls and keeping values in CPU registers across relational operators, this approach delivers great performance, especially for complex expressions and queries [5]. Additionally, they have a fundamental advantage, as they can, in principle, inline and fuse machine code also for user-defined code. However, in both academia and industry, compiling query engines are often regarded as difficult to understand, debug, and profile. Thus, many modern query engines opt for vectorized query execution [1]. We argue, however, that query compilation can be both understandable and maintainable with appropriate design decisions and tooling. With this paper, we want to demonstrate this for our own, open-source compiling query engine LingoDB [3]. LingoDB shares many properties with pioneering compiling systems like Hyper [4] and Umbra [8] but also differs in three main aspects that improve understandability and maintainability:

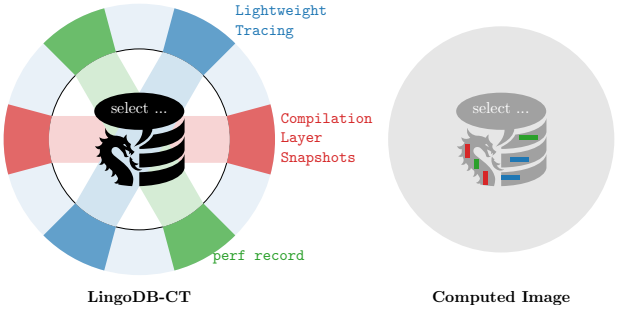



**Figure 1: Like in computer-tomography, LingoDB-CT collects information from different angles about the compilation and execution of a query. Later, this information is correlated to form a more comprehensive picture.**

(1) LingoDB employs a form of layered query compilation with explicit layers (at the cost of small latency increases).
(2) It builds on top of MLIR [6], a *high-level* compiler framework, in addition to using LLVM for machine code generation.
(3) Query optimization is implemented as a set of compiler passes, not as a separate component.

First, using a layered approach allows for examining the compilation at multiple levels of abstraction. Second, using MLIR not only saves a lot of reimplementation effort, but also allows us to build on top of existing mechanisms, e.g., to track provenance of operations in the form of source information. Finally, since query optimization is also part of the compilation stack, we can leverage the same tooling to understand the transformations performed as part of query optimization, which are often hard to understand, even in non-compiling query engines.

To leverage these *architectural* advantages in practice, we developed *LingoDB-CT*, an instrumentation and visualization framework for LingoDB to record and visualize its behavior in both the query optimization and compilation phase, as well as the query execution phase. As sketched in Figure 1, it works for LingoDB similarly as a computer-tomograph (CT) works for scanning human bodies. First, the behaviour of the subject under test (in our case LingoDB during query processing) is recorded from different angles (in our case: different metrics, traces, and profiles). Afterward, the recorded data is *correlated* and assembled to get a picture of the subject's inside.

In this demonstration, we will showcase LingoDB-CT's capabilities for two scenarios: (1) Quickly understanding how LingoDB optimizes and compiles SQL queries. (2) Investigating performance issues by looking at the same time at query optimization, query compilation, and query execution. Both scenarios will be fully interactive, relying on publicly hosted web applications.


*SIGMOD-Companion '25, Berlin, Germany*

ACM ISBN 979-8-4007-1564-8/2025/06
https://doi.org/10.1145/3722212.3725111

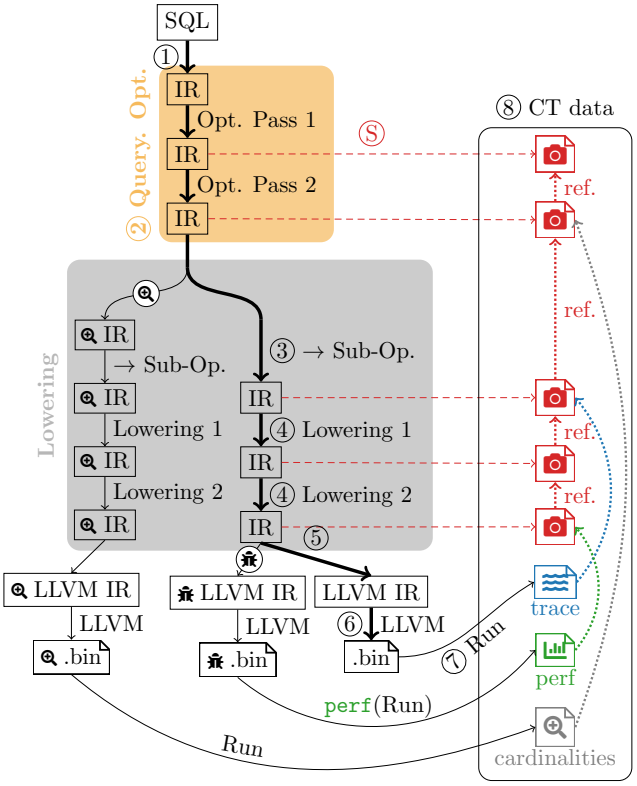



**Figure 2: In addition to the normal optimization and compilation process of LingoDB (thick lines), LingoDB-CT adds several instrumentations and runs to collect a set of correlated (dotted lines) CT data.**

## 2 Demonstration System Overview

LingoDB is a modern, compiling query engine built on top of the high-level compiler framework MLIR [6]. MLIR aims to simplify the development of layered, domain-specific compilers by offering a generic framework that can be extended with custom types, operations and compiler passes. As shown in Figure 2, LingoDB first parses queries ① into a high-level intermediate representation (IR) based on MLIR. Next, LingoDB performs query optimization ② by applying a set of compiler passes to the IR to perform pattern-based simplifications as well as cost-based operator reordering and unnesting of correlated sub-queries. The relational operators (i.e., corresponding to a physical plan) are then ③ lowered to declarative sub-operators [2] that are simpler but enable high-level optimizations such as auto-parallelization of queries. Afterward, additional lowering passes ④ transform sub-operators into imperative operations that are progressively lowered until a low-level IR is reached. Then, LLVM IR is produced from the low-level IR ⑤, which is then compiled to machine code using LLVM ⑥ and finally executed.

Building on top of MLIR offers two main advantages that, by design, help with understanding the optimization and compilation process: (1) MLIR supports dumping the current IR at *any* stage of the compilation pipeline into a human-readable textual serialization format. Additionally, the text format can be reparsed into IR, and compilation can be resumed. (2) By enforcing location information for operations, MLIR paves the way for correlating operations across different stages of the compilation pipeline.

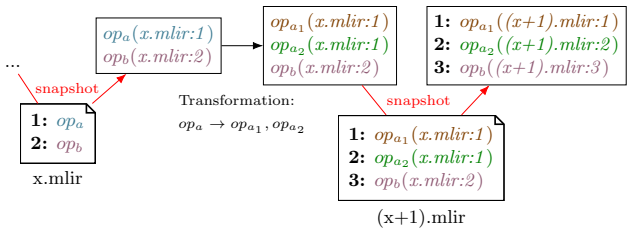



**Figure 3: Snapshotting Process**

### 2.1 Instrumentation

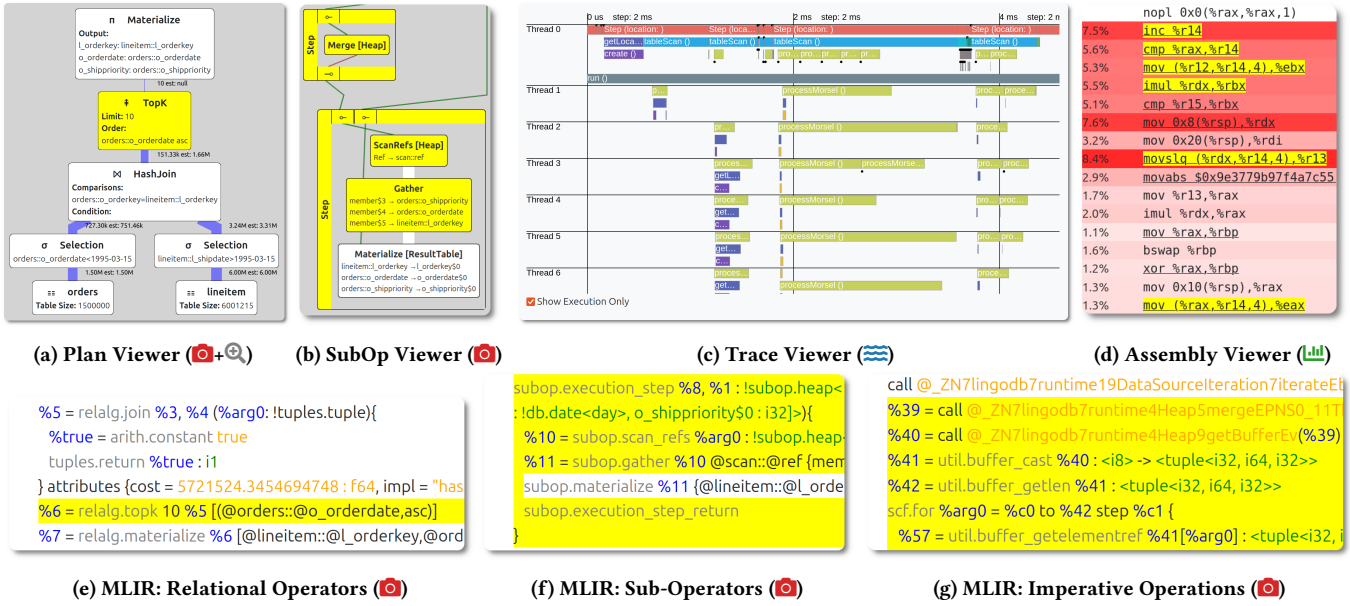
As sketched in Figure 2, LingoDB-CT collects ⑧ different kinds of data that can later be post-processed and visualized. Some of the collected data give more insight into the optimization and compilation process of LingoDB (+). On the other hand, other data (+) allows for insight into the performance characteristics of the actual query execution. However, because of our setup, all collected data is correlated, thus allowing us to gain a unique, unified overview of query optimization, query compilation and query execution.

*2.1.1* *Snapshots of MLIR Modules.* The most critical instrumentation performs *snapshots* of the MLIR module after every compiler pass throughout the optimization and lowering phase Ⓢ. For each snapshot, the MLIR module is written to a new text file, including the current location information (e.g., source file, line) attached to each operation. Next, the new text file's name and line numbers are used to update the location information for all operations.

Figure 3 sketches this for a $(x+1)$th pass rewriting $op_a$ into $op_{a_1}$ and $op_{a_2}$ Before the pass, the operations are tagged with location information pointing to the last snapshot file `x.mlir`. These locations remain valid and are also tracked for rewritings until the next snapshot that writes to `x+1.mlir`.

*2.1.2* *Extracting actual cardinalities.* Generated query plans can often be inefficient due to mismatches between estimated and actual cardinalities. Thus, examining actual cardinalities is important for profiling queries. We extract the actual cardinalities by taking the IR corresponding to the physical plan, instrumenting it with *cardinality operators*, and executing it. This way, we can avoid any overhead for query processing and bloat in the main codebase, but still extract cardinalities that can be matched to the original IR.

*2.1.3* *Execution backend for Profiling.* For LingoDB-CT, we added a profiling execution backend to LingoDB based on LLVM and perf. This backend adds debug information to the generated LLVM IR based on the MLIR location information. Then, LLVM compiles the IR into a dynamic library with debug info, which is loaded into the current process. Before executing the generated code, `perf record` is started to profile the current process and is stopped after the execution finishes. The obtained perf trace is post-processed




(a) Plan Viewer (+) (b) SubOp Viewer () (c) Trace Viewer () (d) Assembly Viewer ()

(e) MLIR: Relational Operators () (f) MLIR: Sub-Operators () (g) MLIR: Imperative Operations ()

**Figure 4: Frontend components with the origin of their data indicated. Selecting an operation in one component (e.g., TopK operator), automatically highlights corresponding operations in other components**

with `perf annotate` to obtain assembly instructions with the number of perf events and the source location, and `perf report` to create a summary showing where time is spent globally.

*2.1.4 Lightweight Traces.* The previous instrumentations allow for gaining overview of the compilation process and identifying performance bottlenecks. To further gain a temporal understanding of the query execution, we added a lightweight tracing mechanism to LingoDB. By following a design originally implemented in Umbra, we can avoid overheads by materializing events into thread-local buffers without contention. After the query was executed, all events (containing start timestamp, duration, event type, and optional metadata) are written to a JSON file ⑦.

## 2.2 Frontend Components

Collecting metrics, as just discussed, is necessary, but not sufficient for gaining useful insights, as visualization matters a lot in practice. In our experience good visualizations abstract away from details while allowing users to dive deeper if necessary, and allow for correlating different views to investigating the same aspect in different views. Thus, we implemented custom components that allow for selecting operations in one component and highlighting related operations in others.

*Plan Visualizations.* Although MLIR's text format is human-readable, it quickly becomes verbose and hard to understand, especially for higher-level representations. We thus implemented a component shown in Figure 4a that renders IR snapshots  corresponding to a physical plan into an easily understandable visual representation. Operators are represented by white nodes containing operator names and additional information such as table names or predicates. The width of edges connecting the operators roughly resembles the estimated cardinalities of tuples following this edge. If actual cardinalities  are known, an additional, half-transparent, differently colored edge is added to quickly show discrepancies in the estimates. Additionally, we developed a slight variant of this component for the Sub-Operator Layer as displayed in Figure 4b.

*Trace Viewer.* To visualize the temporal aspect of execution, we built a custom trace viewer component to display the collected traces  as shown in Figure 4c. Besides scrolling and zooming, it also supports clicking on events to select corresponding operations.

*MLIR Source Viewer.* Existing syntax highlighting libraries do not fully work for MLIR, as MLIR's syntax is operation-specific. We thus built a custom component that renders pre-tokenized MLIR modules that are produced ahead of time from the snapshots  and allows for selecting and highlighting operations.

*Perf Assembly Viewer.* Finally, we implemented a rendering of annotated assembly code  for generated code as shown in Figure 4d. It quickly shows the bottlenecks by using a darker background color for busy instructions, and again supports clicking on and highlighting of instructions.

# 3 Demonstration Proposal

This section walks through the demonstration of LingoDB-CT with the help of Alice and Bob, two imaginary users. Alice is interested in LingoDB and wants to understand (*Scenario 1*) how LingoDB works under the hood for certain queries (e.g., to onboard herself as a developer or evaluate it for advanced use cases). Bob is a system developer working on LingoDB who frequently needs to investigate performance issues.

### 3.1 Scenario 1: Understanding LingoDB

In order to understand how LingoDB works, Alice visits the publicly hosted LingoDB web interface at lingo-db.com/interface. Initially, the web interface displays a SQL editor, as well as buttons for executing the SQL, selecting a dataset (one of TPC-H, TPC-DS, JOB, educational sample database), and populating the SQL editor with a benchmark query. Alice can now explore how LingoDB works under the hood interactively, without needing to install anything:

(1) Alice enters a SQL query or selects one of the benchmarking queries. Optionally, she can also tick a checkbox to retrieve actual cardinalities.
(2) After pressing the `Execute Query` button, the query is executed on a remote virtual machine and the query result with execution and compilation times is displayed below.
(3) If Alice wants to know more about the optimization and compilation process, she can now click on the `Query Plan` tab. This invokes an additional run on the remote machine with some of the instrumentations described above. After a few seconds, the query plan is rendered, as shown in Figure 4a.
(4) Alice is now interested in how one of the operators is implemented (e.g., a TopK operator). She clicks on the operator which then turns yellow.
(5) Now, she can click on other tabs to view the Sub-Operator plan or the textual MLIR representations where the operations corresponding to the selected TopK operator are highlighted, as shown in Figure 4b and Figure 4e,4f,4g. This way, she can now understand how the TopK operator is implemented.

Of course, Alice can also click on lower-level operations to investigate which higher-level operations (e.g., relational operator) they belong to.

### 3.2 Scenario 2: Investigating Performance Issues in LingoDB

For the second scenario, we assume that Bob has been assigned a slow query and needs to identify the underlying performance problem. He first runs a Python script that executes the query with snapshotting and instrumentation enabled to generate a `ct.json` file. Next, Bob opens the LingoDB-CT web application hosted at ct.lingo-db.com and uploads the file in the browser.

(1) Bob first looks at the displayed trace (Figure 4c) to quickly identify if (1) all worker threads are busy and (2) which execution steps dominate the total runtime.
(2) He then starts investigating suspicious execution steps that dominate the runtime or are taking longer than expected, by clicking on them in the trace viewer.
(3) This highlights the corresponding operators in the query plan rendered in the bottom left. Bob can now take a closer look at this operator:
    (a) Did the query optimizer do a good job here? Or should the operators have been ordered differently due to incorrectly estimated cardinalities?
    (b) How expensive is the operator supposed to be? Are there any potentially expensive expressions inside the operator?
(4) If the root cause for the poor performance is still unclear, Bob can now click on `ASM` to investigate the generated machine code annotated by `perf`. Instructions belonging to the current operator are again highlighted as shown in Figure 4d.
(5) For a more detailed analysis, Bob can also switch to the sub-operator view and examine the generated machine code for each sub-operator.
(6) If Bob finds an unusually busy instruction, he clicks on it to select it and quickly looks at the other layers to find the corresponding operations.
(7) If there is still no clear explanation, Bob can also look at the integrated `perf` summary which lists (kernel) functions and the percentage of execution time.

Once Bob has found and selected a suspicious operation, he can now trace its origin back by clicking on `Diff`.

(7) The web application then displays two MLIR modules side-by-side. On the right, the IR containing the selected operation (highlighted) is displayed. On the left, the snapshot taken before the one on the right is shown, with the corresponding operations being highlighted.
(8) Bob looks at both highlighted regions. If the left side is already not as expected, Bob continues navigating to the previous snapshot while the corresponding operations remain highlighted.
(9) Once he notices a meaningful change (i.e., left side is correct, right suboptimal), he has identified a sub-optimal pass that can be debugged in isolation.

## Acknowledgments

We thank Google for their support in the form of a PhD Fellowship. We also thank Tobias Schmidt, Maximilian Reif, and Altan Birler for their valuable feedback.

## References

[1] Peter A. Boncz, Marcin Zukowski, and Niels Nes. 2005. MonetDB/X100: Hyper-Pipelining Query Execution. In *CIDR*. www.cidrdb.org, 225–237.
[2] Michael Jungmair and Jana Giceva. 2023. Declarative Sub-Operators for Universal Data Processing. *Proc. VLDB Endow.* 16, 11 (2023), 3461–3474.
[3] Michael Jungmair, André Kohn, and Jana Giceva. 2022. Designing an Open Framework for Query Optimization and Compilation. *Proc. VLDB Endow.* 15, 11 (2022), 2389–2401.
[4] Alfons Kemper and Thomas Neumann. 2011. HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots. In *ICDE*. IEEE Computer Society, 195–206.
[5] Timo Kersten, Viktor Leis, Alfons Kemper, Thomas Neumann, Andrew Pavlo, and Peter A. Boncz. 2018. Everything You Always Wanted to Know About Compiled and Vectorized Queries But Were Afraid to Ask. *Proc. VLDB Endow.* 11, 13 (2018), 2209–2222.
[6] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques A. Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. 2021. MLIR: Scaling Compiler Infrastructure for Domain Specific Computation. In *CGO*. IEEE, 2–14.
[7] Thomas Neumann. 2011. Efficiently Compiling Efficient Query Plans for Modern Hardware. *Proc. VLDB Endow.* 4, 9 (2011), 539–550.
[8] Thomas Neumann and Michael J. Freitag. 2020. Umbra: A Disk-Based System with In-Memory Performance. In *CIDR*. www.cidrdb.org.