UmbraPerf - Profiling Results Tailored for DBMS Developers

Alexander Beischl
Technical University of Munich
Munich, Germany
beischl@in.tum.de

Thomas Neumann
Technical University of Munich
Munich, Germany
neumann@in.tum.de

ABSTRACT

Developing a code-generating Database Management System requires tight profiling and performance-tuning iterations. However, existing profilers report results at instruction or function level, making it challenging to correlate them with constructs like query plan operators to derive actionable insights.

In this demonstration, we show how to solve this issue, building on our previous work, Tailored Profiling. We introduce UmbraPerf, a novel profiling tool that combines profiling samples with metadata to map profiling results to a DBMS's internal abstractions, from query plan components down to generated code. Its interactive frontend visualizes data across multiple abstraction levels, enabling developers to narrow performance bottlenecks from operators down to individual instructions. We demonstrate its utility through two scenarios using our publicly available frontend.

PVLDB Reference Format:
Alexander Beischl and Thomas Neumann. UmbraPerf - Profiling Results Tailored for DBMS Developers. PVLDB, 18(12): 5291 - 5294, 2025.
doi:10.14778/3750601.3750654

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://umbraperf.github.io/umbraperf.

1 INTRODUCTION

Code-generating Database Management Systems (DBMSs) aim to produce highly optimized code for query execution. Consequently, developers of such systems invest substantial effort in fine-tuning the system. To identify performance bottlenecks and tuning opportunities, they measure the DBMS with profiling tools [1, 3, 5, 10]. However, interpreting profiling results is challenging because the link between the query plan and executed code is lost. Ideally, developers want profiling results mapped to their domain's familiar abstractions, e.g., query plan operators and pipelines for a DBMS, to facilitate rapid optimization iterations.

Most existing profiling tools, like *Linux perf, hotspot* or *vTune*, only provide results on assembly instruction-, source line- or function-granularity [1, 3, 10]. For traditional DBMSs, e.g., Postgres, experienced developers can map most of the results back to the system's components because the executed code belongs to the DBMS itself, with which they are familiar. Still, profiling results of shared code locations, e.g., a hash table implementation used in multiple operators like *hash-join* or *group by*, remains challenging, as they cannot be easily disambiguated. For code-generating DBMSs, this is even more challenging as they generate machine code for each query execution, and the connection between the system components and generated code is lost during compile time. Furthermore, advanced optimizations like operator fusion obfuscate boundaries as they interleave multiple operators in a single tight loop of generated code, making it exceedingly cumbersome to link profiling results back to the corresponding query plan components.

We addressed this issue in our previous work with *Tailored Profiling* to map generated code to system components on multiple abstraction levels [2]. Building on this, we introduce *UmbraPerf*, a tool that collects and visualizes profiling data on abstractions familiar to developers of code-generating DBMSs like Umbra [9], demonstrating the benefits of domain-specific profiling solutions.

UmbraPerf correlates profiling results at the individual sample level with DBMS components, i.e., operators, pipelines, and generated IR instructions, even for multiple events and shared code locations. To our knowledge, no existing profiler offers this capability. UmbraPerf achieves this by preserving the link between system components, generated code, and the profiled executed code.

UmbraPerf collects fine-grained profiling data, sampling query execution at ≈0.2-1MHz, and correlates them with DBMS components to provide detailed insights, e.g., each operator's over-time activity, memory access- or cache-miss patterns. Our tool's interactive frontend processes this data on the fly, automatically aggregating and visualizing profiling results at multiple abstraction levels from high-level, e.g., query plan operators, down to generated instructions. Developers can customize the reported results in the frontend, filtering for operators, pipelines, or specific time intervals. Thereby, UmbraPerf enables rapid detection of optimization opportunities and seamless transitions from broad overviews to precise investigations of performance bottlenecks or validation of applied optimizations.

We demonstrate this by showcasing UmbraPerf's capabilities in two scenarios for the DBMS Umbra: (1) Pinpointing a performance issue caused by L3-cache misses. (2) Evaluating the performance optimization after introducing a *group-join*. Both scenarios are interactive and available on our publicly hosted web application.

2 SOLUTION OVERVIEW

UmbraPerf provides profiling results for code-generating DBMS's query execution tailored to a DBMS's abstractions and can be used interactively and without instrumentation overhead for developers. Our approach is widely applicable to code-generating DBMSs, and we illustrate its integration using our DBMS Umbra as an example.

UmbraPerf is separated into two components (cf. Figure 1). The **backend**, integrated into the query processing engine, tracks the links between the system components and generated code, collects

Proceedings of the VLDB Endowment, Vol. 18, No. 12 ISSN 2150-8097.
doi:10.14778/3750601.3750654

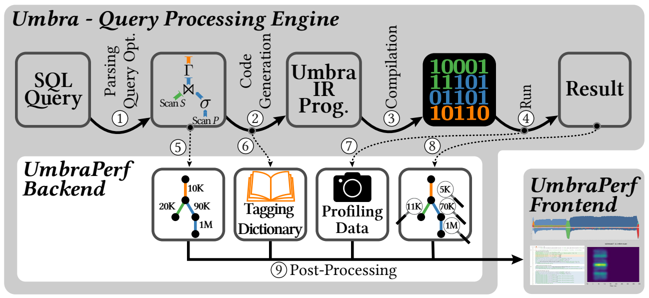


Figure 1: UmbraPerf collects meta-data at different stages of query processing and combines it with profiling data to report performance insights tailored to a DBMS's abstractions.

required meta-data, instruments the profiling, and post-processes all data. The **frontend** visualizes the profiling results in an interactive web application that processes the profiling data and allows developers to filter for the insights they need. To understand how UmbraPerf works in detail, we will first provide a brief overview of Umbra's query processing and then explain UmbraPerf's architecture and integration into the DBMS.

2.1 Query Processing

Umbra [9] is a high-performance relational database system that applies data-centric code generation for query execution. Umbra's query processing engine (cf. Figure 1), written in C++, ① parses a SQL query into a query plan of relational operators and performs query optimization through multiple optimization passes on the plan. ② Next, Umbra generates Umbra IR code (a custom subset of LLVM IR) for the optimized query plan. ③ Eventually, we use the LLVM compiler framework [7] to compile Umbra IR into optimized native machine code, and ④ execute it to run the query.

Operator Fusion. Since Umbra applies data-centric code generation, it splits the query plan into tuple materialization points and generates code pipeline-wise between materialization points, producing tight loops of Umbra IR instructions with *operator fusion* [8]. For the simplified query in Figure 1, the scan over P, the probe side of the $\bowtie$, and materialization in the Γ form a pipeline. During code generation, Umbra fuses the instructions generated for the operators in this pipeline into a tight loop, blurring operator boundaries.

While this execution model produces highly optimized query-specific code, keeping values in CPU registers as long as possible for excellent data locality (due to operator fusion) and utilizing the available computing resources, it makes profiling and debugging difficult. Since Umbra's query processing engine is split into *compile time*, performing all steps until compiling the generated code, and *runtime*, actually executing the query with machine-generated code, the link between the DBMS's abstractions (i.e., query plan) and the profiled runtime code is lost. Thus, existing profilers fail to map the profiling results to the system's components.

2.2 Profiling Backend

UmbraPerf's backend collects meta-data at different stages of the query processing engine, as shown in Figure 1.

Query Plan and Cardinalities. DBMS developers use the query plan with pipelines, operators, and their (estimated) cardinalities as central abstraction to reason about optimizations of their system's query execution. Thus, UmbraPerf extracts the query plan with cardinality estimates ⑤ after the query optimization phase and the actual cardinalities ⑧ after the query execution.

Tagging Dictionary. During code compilation, we track the connection between query plan pipelines and operators to their generated Umbra IR instructions with *Abstraction Tracking* and add *Register Tagging* to disambiguate shared code locations [2]. Whenever an Umbra IR instruction is generated, we add a mapping from the instruction to its responsible operator and pipeline to the ⑥ *Tagging Dictionary* [2]. Therefore, we create unique IDs for each operator and pipeline of the query plan. Since Umbra IR follows SSA-form, we identify generated instructions reusing their Umbra IR variables as unique IDs. At the end of code generation, we write the Tagging Dictionary into a meta-data file, which is used during post-processing to link the samples back to operators and pipelines.

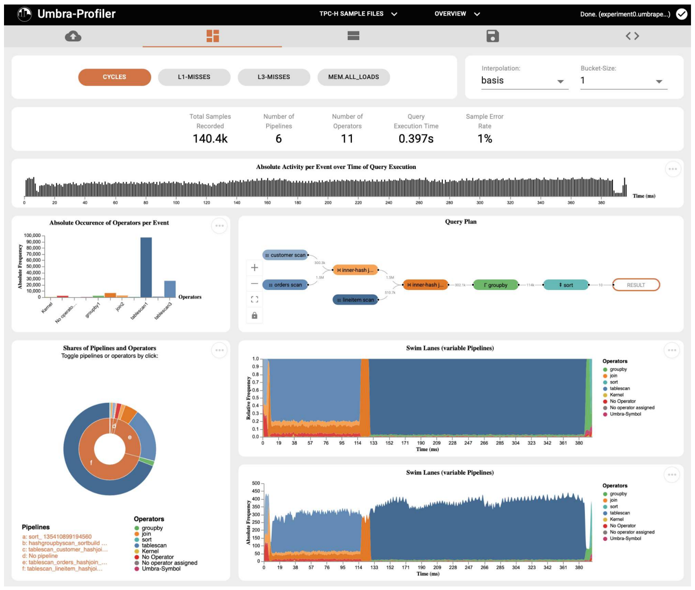
Profiling Query Execution. UmbraPerf uses `perf record` to ⑦ profile the query execution. Before running the generated code for query execution, UmbraPerf starts `perf record`, attaches it to the query execution's process, and stops it after the execution finishes. To reduce latency and only record the query execution, we added interprocess communication to `perf`. After starting `perf record`, UmbraPerf waits for `SIGUSR1`, indicating `perf record` is ready, and then starts the query execution. After query execution, we send a `SIGKILL` to perf, notifying it to end profiling and wrap up the `perf.data` file. Leveraging `perf record`'s capabilities, UmbraPerf samples profiling data with 0.2-1 MHz and captures multiple hardware events, e.g., cycles, cache-, branch-misses, memory accesses, thereby enabling developers to investigate potential correlations at the level of individual operators.

Postprocessing. After the query execution is finished, UmbraPerf produces an `.umbraperf`-file, which contains the profiling data augmented to the query plan abstractions. Therefore, we use `perf script` to retrieve the individual samples with precise timestamps and their recorded data from the `perf.data`-file. Since Umbra uses LLVM to compile Umbra IR into machine code, `perf script` can utilize the debug info generated by LLVM to resolve native instructions to Umbra IR instructions. Then, we process the individual samples and link them to pipelines and operators with the Tagging Dictionary and query plan, producing a `.umbraPerf`-file that contains samples linked to the DBMS abstractions.

2.3 Profiling Frontend

While fine-grained profiling data is critical for detailed performance insights, its practical value depends on how effectively developers can interpret and act upon it.

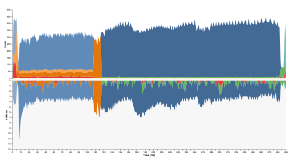
Performance Considerations. The performance of development tools is often undervalued despite their potential to accelerate optimization iterations. UmbraPerf addresses this issue through a web application frontend, which processes the augmented profiling samples of the `.umbraPerf`-file in real time using Rust compiled to WebAssembly, and visualizes them in interactively customizable views



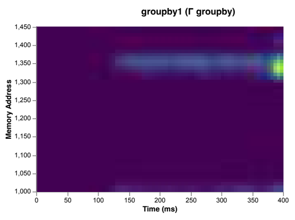
Interpolation: basis
Bucket-Size: 1

Total Samples Recorded 140.4k
Number of Pipelines 6
Number of Operators 11
Query Execution Time 0.397s
Sample Error Rate 1%

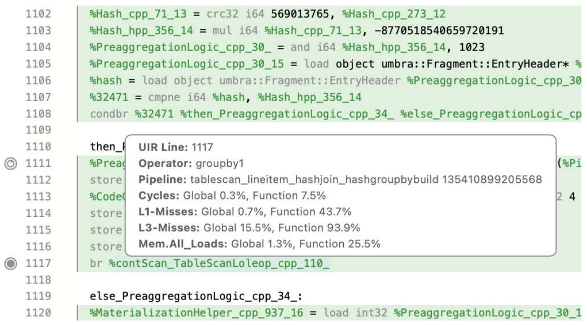
(a) Overview View



(b) Event Correlation View



(c) Memory Access View



```
1102  %Hash_cpp_71_13 = crc32 i64 569013765, %Hash_cpp_273_12
1103  %Hash_hpp_356_14 = mul i64 %Hash_cpp_71_13, -8770518540659720191
1104  %PreaggregationLogic_cpp_30_ = and i64 %Hash_hpp_356_14, 1023
1105  %PreaggregationLogic_cpp_30_15 = load object umbra::Fragment::EntryHeader* %
1106  %hash = load object umbra::Fragment::EntryHeader %PreaggregationLogic_cpp_30
1107  %32471 = cmpne i64 %hash, %Hash_hpp_356_14
1108  condbr %32471 %then_PreaggregationLogic_cpp_34_ %else_PreaggregationLogic_cp
1109
1110  then_
1111  %Preag                                                                    (%Pi
1112  store
1113  %Code(                                                                   2 4
1114  store
1115  store
1116  store
1117  br %contScan_TableScanLoleop_cpp_110_
1118
1119  else_PreaggregationLogic_cpp_34_:
1120  %MaterializationHelper_cpp_937_16 = load int32 %PreaggregationLogic_cpp_30_1
```

UIR Line: 1117
Operator: groupby1
Pipeline: tablescan_lineitem_hashjoin_hashgroupbybuild 135410899205568
Cycles: Global 0.3%, Function 7.5%
L1-Misses: Global 0.7%, Function 43.7%
L3-Misses: Global 15.5%, Function 93.9%
Mem.All_Loads: Global 1.3%, Function 25.5%

(d) Umbra IR Report

Figure 2: UmbraPerf's frontend visualizes profiling results for different abstractions. It supports filtering by time interval, operator, and pipeline, with synchronized views across all components. Operators are consistently color-coded in every view.

built on Vega [6]. Performing client-side data processing, UmbraPerf minimizes interaction latencies and supports filtering profiling data for multiple parameters, streamlining the developer's workflow.

Design Considerations. Choosing the right level of abstraction is important to avoid overwhelming developers, but present the results in a familiar way while still not hiding details. Our tool assists developers in their analysis by providing four different views (*Overview*, *Event Correlation*, *Memory Access Analysis*, *Umbra IR report*), each offering different levels of abstraction, visualizations, and functionalities. Through these views, developers can transition from high- to low-level abstractions, seamlessly exploring more details to pinpoint potential bottlenecks more efficiently. They can select results for different profiling events and visualize them in parallel to identify correlations. UmbraPerf also supports interactive filtering for operators, pipelines, and time intervals to narrow performance investigation from the entire query execution to specific time frames or specific operators. For a seamless transition between the different abstraction levels, our tool synchronizes filtering selections between all views and computes them in the background.

Overview (Figure 2a). This view is the starting point for performance analysis and provides a high-level overview of the query execution: which operators were involved, ran in parallel, and how much time did they consume. A header displays key statistics, e.g., query execution time, sample counts, and developers can choose among recorded events to switch the visualizations to a specific event.

Timeline: Our tool displays the number of recorded samples over time and supports time-interval-based filtering to focus on a specific segment of the query execution.

Operator Plan: Since query optimization relies on cardinality estimates, comparing estimated and actual cardinalities is crucial for assessing each operator's impact and identifying prediction mismatches. Accordingly, UmbraPerf visualizes the physical query plan by representing operators as nodes and input dependencies as edges (pipelines), annotated with the actual cardinalities. Hovering over an operator reveals a tooltip with cardinality estimates and the top five Umbra IR instructions for the selected profiling event. Clicking on an operator toggles its inclusion in the performance results and synchronizes these changes across all visualizations.

Operator & Pipeline Costs: A bar plot displays each operator's performance impact or share of the selected profiling event. A sunburst chart represents the corresponding data for pipelines in its inner ring and operators of the respective pipeline in the outer ring. Together, these visualizations enable an assessment of both individual operator and entire pipeline impacts. Clicking on an operator/pipeline toggles its inclusion in the results.

Swimlanes: visualize operator activity over execution time for a selected profiling event in relative (percentage) and absolute (sample count) measures. Swimlanes intuitively show the temporal performance and inter-operator interactions, facilitating the detection of unexpected behavior in specific query execution segments.

Event Correlation (Figure 2b). To further investigate unexpected operator behavior, we compare results across different profiling events using a dual swimlane view. This visualization facilitates the identification of correlations and the isolation of potential causes, e.g., a cache-miss peak for an operator during a segment with low activity. To convey the context and enable selection, this view also contains the header, timeline, operator plan, and sunburst graph.

Memory Access Analysis (Figure 2c). Since multiple operators are executed simultaneously (operator fusion), distinguishing among them is crucial for analyzing memory access patterns. UmbraPerf addresses this by visualizing each operator's memory accesses as an X-Ray plot, with the y-axis representing memory addresses and the x-axis execution time. Developers can choose between each operator's accessed memory addresses or deltas between consecutive accesses to detect random access patterns.

Umbra IR Report (Figure 2d). UmbraPerf provides an augmented `perf report`-style view for the generated Umbra IR instructions, the lowest abstraction level in Umbra. Each instruction is annotated with its operator, pipeline, and performance metrics for the selected profiling event, while hovering over it reveals metrics for all events. Additionally, developers can apply the same filtering options as for the other components to refine their analysis. With the Umbra IR Report, developers can pinpoint performance bottlenecks to the level of generated instructions while preserving the overall context of query execution.

3 DEMONSTRATION PROPOSAL

In our demonstration, participants are invited to analyze Umbra's query execution using our publicly hosted interactive UmbraPerf frontend [1] (cf. Figure 2) and investigate profiling results for potential performance bottlenecks. We provide pre-uploaded `.umbraperf`-files for all TPC-H queries and our example scenarios. Visitors can also run their own queries (for TPC-H, TPC-DS, JOB) on one of our machines using Umbra and analyze the query execution with UmbraPerf to experience the profiling workflow in practice.

To illustrate the workflow of UmbraPerf, we walk through two example scenarios in this section with the help of DBMS developer Alice. *Scenario 1:* Alice investigates TPC-H Query 3 for optimization opportunities. *Scenario 2:* After introducing a *group-join*, Alice revisits TPC-H Query 3 to assess the improvements achieved.

3.1 Scenario 1: Identifying Optimization Opportunities

Alice aims to optimize a slow query by analyzing its execution to identify performance issues. First, Alice runs the query in Umbra and profiles it using the UmbraPerf mode (CMakeProfile), producing a `.umbraperf` file. Next, Alice uploads the file to the frontend to begin the analysis:

(1) Alice reviews the *Overview's* query plan and swimlanes (cf. Figure 2a) to determine which operators dominate execution and detect anomalies.
(2) Selecting different events, Alice observes that *join 1* and the *group by* operator cause a peak in L3-cache misses.
(3) Alice switches to the *Event Correlation* view (cf. Figure 2b) to assess the impact of these cache misses and correlate them with temporal operator activity, noting that the *group by* exhibits frequent cache misses despite limited activity.
(4) To investigate further, Alice examines the memory access addresses for the *group by* in the *Memory Access Analysis* view (cf. Figure 2c). By comparing these addresses with those of other operators, Alice identifies that the L3-cache misses originate at the same addresses as those in *join 1*.
(5) Finally, Alice accesses the *Umbra IR Report* (cf. Figure2d), selects L3-cache misses as the event, and filters for *join 1*, *group by*, and the time interval corresponding to the cache-miss peak. At the UmbraIR instruction level, Alice discovers that the cache misses result from building the hash table for `l_orderkey` in *join 1* and constructing a grouping hash table for `l_orderkey` in the *group by*.

[1] Available at https://umbraperf.github.io/umbraperf

3.2 Scenario 2: Evaluating an Optimization

After identifying the performance issue in Scenario 1, Alice implemented a *group-join* [4] to avoid building the hash table for `l_orderkey` twice in consecutive operators. To validate the effectiveness of the *group-join*, Alice profiles the query again and compares the new results in a second browser window.

(1) The comparison reveals that query execution is 14% faster, with L3-cache misses occurring only during the single hash table build within the *group-join* as expected.
(2) Now Alice can start fine-tuning the new operator by examining the Umbra IR Report to assess the cost and metrics of each generated instruction for the new operator.

4 CONCLUSION

In this work, we introduce UmbraPerf, a profiling tool that bridges the gap between low-level profiling data and the system abstractions of code-generating DBMSs. UmbraPerf correlates profiling results with DBMS abstractions, such as query plan operators. Its interactive frontend presents profiling results at developer-familiar abstractions, facilitating rapid performance-tuning iterations, as demonstrated in our scenarios. While we have integrated UmbraPerf into Umbra, its design is widely applicable to other code-generating DBMSs.

ACKNOWLEDGMENTS

We thank Johann-Matthias Schworm and Michael Schott for their contributions to UmbraPerf as part of their master's theses and André Kohn for his co-supervision.

REFERENCES

[1] 2025. *Linux perf.* Retrieved 2025-05-26 from https://github.com/torvalds/linux/tree/master/tools/perf
[2] Alexander Beischl, Timo Kersten, Maximilian Bandle, Jana Giceva, and Thomas Neumann. 2021. Profiling dataflow systems on multiple abstraction levels. In *EuroSys '21: Sixteenth European Conference on Computer Systems, Online Event, United Kingdom, April 26-28, 2021*, Antonio Barbalace, Pramod Bhatotia, Lorenzo Alvisi, and Cristian Cadar (Eds.). ACM, 474–489. https://doi.org/10.1145/3447786.3456254
[3] Intel Corporation. 2025. *Intel VTune Profiler.* Retrieved 2025-05-26 from https://software.intel.com/en-us/vtune
[4] Philipp Fent and Thomas Neumann. 2021. A Practical Approach to Groupjoin and Nested Aggregates. *Proc. VLDB Endow.* 14, 11 (2021), 2383–2396. https://doi.org/10.14778/3476249.3476288
[5] Brendan D. Gregg. 2014. *Flame Graphs.* Retrieved 2025-05-26 from http://www.brendangregg.com/flamegraphs.html
[6] UW Interactive Data Lab. 2025. *Vega – A Visualization Grammar.* Retrieved 2025-05-26 from https://vega.github.io/vega/
[7] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation. In *CGO.* 75–88.
[8] Thomas Neumann. 2011. Efficiently Compiling Efficient Query Plans for Modern Hardware. *Proc. VLDB Endow.* 4, 9 (2011), 539–550.
[9] Thomas Neumann and Michael J. Freitag. 2020. Umbra: A Disk-Based System with In-Memory Performance. In *CIDR 2020, Amsterdam, The Netherlands, January 12-15, 2020, Online Proceedings.*
[10] Milian Wolff. 2016. *Hotspot - the Linux perf GUI for performance analysis.* Retrieved 2025-05-26 from https://github.com/KDAB/hotspot