SHORT-PAPER

UDFBench: A Tool for Benchmarking UDF Queries on SQL Engines

YANNIS FOUFOULAS, Athena - Research and Innovation Center in Information, Communication and Knowledge Technologies, Athens, Attica, Greece

THEONI PALAIOLOGOU, National and Kapodistrian University of Athens, Athens, Attica, Greece

ALKIS SIMITSIS, Athena - Research and Innovation Center in Information, Communication and Knowledge Technologies, Athens, Attica, Greece

Open Access Support provided by:

Athena - Research and Innovation Center in Information, Communication and Knowledge Technologies

National and Kapodistrian University of Athens

Published: 22 June 2025

Citation in BibTeX format

SIGMOD/PODS '25: International Conference on Management of Data
June 22 - 27, 2025
Berlin, Germany

Conference Sponsors:
SIGMOD

https://doi.org/10.1145/3722212.3725139
ISBN: 9798400715648

UDFBench: A Tool for Benchmarking UDF Queries on SQL Engines

Yannis Foufoulas
Athena Research Center
Greece
johnfouf@di.uoa.gr

Theoni Palaiologou
University of Athens and Athena R.C.
Greece
cs2210019@di.uoa.gr

Alkis Simitsis
Athena Research Center
Greece
alkis@athenarc.gr

Abstract

User-defined functions (UDFs) extend SQL with functional capabilities. However, integrating UDFs efficiently into a data engine is a challenge, mainly due to the mismatch of the UDF execution and SQL processing environments. Several techniques have been proposed to deal with this challenge spanning a variety of logical and physical optimizations, and architecture designs. These however are not evident to the casual user, who would benefit from knowing the pros and cons of the available solutions. Towards this direction, we present UDFBench, a benchmarking toolkit for running seamlessly UDF queries (queries with UDFs) on a variety of SQL engines and presenting to the user execution results accompanied with highlights of the main reasons driving the query performance. UDFBench comprises a set of queries and UDFs of varying complexity based on a real-world application and data. Our current implementation has been deployed on four popular database engines with different characteristics. Our pluggable design enables extending UDFBench to additional systems and workloads.

CCS Concepts

• **Information systems → Data management systems**.

Keywords

SQL, user-defined functions, performance benchmarking

ACM Reference Format:
Yannis Foufoulas, Theoni Palaiologou, and Alkis Simitsis. 2025. UDFBench: A Tool for Benchmarking UDF Queries on SQL Engines. 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/10.1145/3722212.3725139

1 Introduction

Database engines extend SQL with user-defined functions (UDFs), which enable implementing arbitrary programs coded in a variety of programming languages such as C/C++, Python, Java, Scala, R, JS, and so on. Several UDF types are typically supported (e.g., scalar, aggregate, table, window), but their execution is routinely suboptimal as it does not fully exploit the engine capabilities for query optimization and execution. This is due to various challenges [4, 5], including: (a) data type compatibility, data conversions, data copies, context switches, (b) execution mode, thread/process parallelism, in/out process execution, tuple/operator at a time execution, (c) suboptimal query optimization of relational operators and UDFs, (d) missed UDF optimization opportunities via parallelization, vectorization, function inlining, loop-fusion, just-in-time compilation, and so on.

Most popular database engines and research prototypes address several of these challenges in their own way, making these systems suitable for some applications and suboptimal for others. However, understanding the capabilities and limitations of each system is a challenge. Database benchmarks such as the TPC benchmarks suite [15] and other popular benchmarks such as the Join Order Benchmark (JOB) [8], OptMark [9], and Star Schema Benchmark (SSB) [10], focus on query processing challenges but do not consider complex UDFs. SQL-ProcBench [7] considers procedural SQL workloads, but it has two limitations. First, it is expressed in three SQL dialects PL/SQL, PL/pgSQL, and T-SQL, hence it is unable to capture the challenges of UDFs coded in popular languages such as C/C++ or Python [11]. Second, it contains 24 scalar and 10 table UDFs invoked mostly by simple queries ('select udf from dual'), whereas its few more complex queries contain a single, simple UDF.

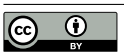
Our work fills a critical gap in database benchmarking by focusing on UDF-centric query processing. UDFBench focuses on measuring and analyzing the overheads and optimizations specific to UDF queries. It also enables investigating the interplay between relational operators and UDFs, identifying optimization opportunities, and evaluating execution environments for diverse UDF types, such as scalar, aggregate, and table UDFs. To achieve these goals, UDFBench comprises a relational schema populated with real-world data, 42 UDFs of various types and complexity, and 21 queries containing a blend of relational and UDF operators, at various nesting levels, ranging from simple queries that highlight UDF overheads to complex queries involving multiple relational and UDF operators that present intriguing optimization opportunities.

UDFBench adopts a modular design for evaluating diverse execution environments, such as row vs. column stores, memory-based vs. disk-based systems, and server-based vs. embedded engines. We demonstrate UDFBench portability and efficacy on four database engines: MonetDB, PostgreSQL, DuckDB, and SQLite. In particular, we showcase the practical utility of UDFBench by presenting:

(a) Execution framework: A user-friendly API packaged as a Python library to execute UDFBench experiments programmatically. Users may import the library, configure the experimental environment, and run experiments with a single command, e.g., `udfbench.experiment.run(query,["monetdb"])`.

(b) Interactive testbed: UDFBench is deployed via Jupyter notebooks, with workflows to run UDFBench queries, visualize performance metrics (e.g., execution time, CPU and memory utilization), provide highlights to explain query performance and generate a comparative analysis across multiple database engines.

(c) Extensibility features: UDFBench supports adding custom UDFs and queries, allowing users to tailor the benchmark to specific workloads. Furthermore, UDFBench's modular design allows



SIGMOD-Companion '25, Berlin, Germany

ACM ISBN 979-8-4007-1564-8/2025/06
https://doi.org/10.1145/3722212.3725139

sub	type	process	return	in → out
1	scalar	1 tuple at a time or a scalar value	one value	tuple → value
2	aggregate	1 group at a time	one value	group → value
3	table	1 tuple at a time	one tuple	tuple → tuple
4	table	1 tuple at a time	one table	tuple → table
5	table	1 table	one table	table → table
6	table	1 group at a time	one tuple	group → tuple
7	table	1 group at a time	one table	group → table
8	table	1 scalar value	one table or tuple	value → table
9	table	1 table	one value usually with a side effect	table → value
10	table	1 table	one tuple	table → tuple

Table 1: UDF type (type) and sub-type (sub) classification

integrating new data engines, hence, extending its applicability to emerging database technologies.

Next, we briefly describe UDFBench and present how it can be used to investigate various scenarios of UDF query execution across a multiplicity of database engines.

2 The UDF Benchmark

Schema. The benchmark schema hails from a real-world application, OpenAIRE (openaire.eu), that promotes open science and is supported by 65 European universities and other institutions. OpenAIRE connects to 1000+ data providers and to date, it has harvested over 130M publications, 2M datasets, 85K research software artifacts, and 1.5M research projects from 23 funders. It uses 150+ UDFs to perform text mining, information extraction, and text analytics over Open Access publications.

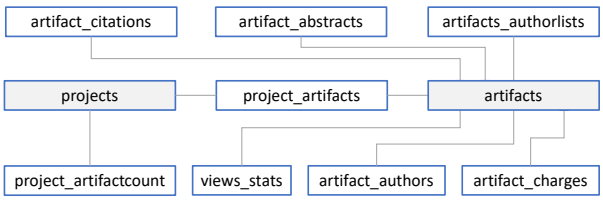


Figure 1: UDFBench: Schema

The benchmark schema (see Figure 1) is a curated subset of the OpenAIRE schema. It comprises two core tables, artifacts and projects, connecting to 7 tables containing author metadata, author lists per artifact (in JSON), artifact abstracts and accessing fees, artifacts related to a project, and 1 table with view statistics for each artifact. Currently, we offer three scale factors: small (13M rows), medium (60M rows), and large (120M rows). These are adequate to measure and evaluate the implications of UDFs in UDF queries. Larger datasets could mingle the UDF overheads (our focus) with other factors such as I/O communication, disk spill issues, etc. However, one may extract a larger snapshot of the OpenAIRE data.

UDFs. The 42 UDFs (24 scalar, 3 aggregate, and 14 table UDFs) are classified into 10 subtypes (sub) according to their input and output as shown in Table 1. Each subtype enables different optimization opportunities. For example, table UDFs that process one tuple at a time (T-3, T-4) are parallelizable using data partitioning, while table UDFs that process the entire table at once (T-5, T-9, T-10) are generally blocking operators. Table UDFs that process one group at a time (T-6, T-7) differ from aggregate UDFs as they do not return a single value, however, they enable optimizations similar to aggregate UDFs. The 42 UDFs have various characteristics, such as they allow pipeline/blocking execution, are parallelizable, are stateful, (may) have side-effects, are implemented in multiple programming languages (e.g., Python, C, SQL).

Queries. The 21 benchmark queries are classified into four classes. (QC1) Queries with UDFs (mostly) and a few relational operators; useful for evaluating UDF related overheads. (QC2) Queries with a blend of UDFs and expensive relational operators (e.g., join, group) or nesting; for evaluating the interaction of UDFs with relational operators. (QC3) Queries with UDFs and complex relational logic; for evaluating the impact of UDFs in query execution/optimization. (QC4) Queries with UDF and DML operations (e.g., insert, update).

The queries capture different characteristics, including: (a) query class (QC1 to QC4), (b) various UDF types, (c) various data types, simple (text, numeric, etc.) or complex (JSON, list, dict, etc.), (d) may process null values, (e) contain varying UDF and query nesting depth, (f) could benefit from UDF optimizations such as just-in-time compilation, (loop) fusion, vectorization, parallelization, (g) could operate in-process, (h) contain dynamic, polymorphic, stateful UDFs, in one or more languages, and (i) contain a varying number of expensive relational operators (e.g., joins, aggregates).

3 Demonstrable Features

The UDFBench architecture is designed to provide a workflow for evaluating and analyzing the performance of UDF queries across diverse database engines. It comprises the following components.

Schema Initialization and Data Loading. This component sets up the experimental testbed by creating and populating the benchmark schema, ensuring that the tested database engines are prepared with identical schemas and data volumes for fair and consistent comparisons. Users choose the dataset scale (small, medium, large) and specify execution parameters, such as the storage location for each database (e.g., HDD, SSD, or memory).

Execution Configuration. Users define the parameters and workload characteristics for the experiment(s). Execution parameters include configurations for running queries on cold (uncached) or warm (cached) datasets, selecting execution modes such as serialized or multi-threaded setup, and deciding between running single queries or configurable batches of multiple queries. Additionally, users specify the metrics to monitor during the experiment, such as latency, CPU usage, memory footprint, I/O operations, and more.

Experiment Executor. This component orchestrates the execution of benchmark queries on one or more database engines. It ensures consistency in how queries are run by abstracting engine-specific details and providing a uniform execution interface. By supporting single-engine and multi-engine experiments, this component facilitates comparative analyses across different database technologies.

Metrics Analysis and Visualization. When an experiment completes, the results are automatically collected, analyzed, and consolidated into insights reports. This component provides: (a) metrics reporting: (aggregated) results for the selected performance metrics; (b) visualization tools: charts of execution time and resource utilization, and lists of notable insights.

Figure 2: Presentation scenarios

These features help users identify performance bottlenecks, compare execution strategies, and explore optimization opportunities.

4 Our Presentation

We present the functionality of UDFBench through various scenarios designed to highlight typical challenges in UDF queries.

Scenario 1 - UDF overheads. The first scenario utilizes 3 simple UDF queries Q1, Q2, and Q3 from query class QC1. The first extracts the day, month, and year as integers from a date field using three Python scalar UDFs, respectively. The second performs the same functionality with one Python table UDF. The third is same as Q1 but the 3 UDFs are coded in three languages: Python, C, and SQL.

For UI, we employ Jupyter notebooks (jupyter.org). The user imports the udfbench package and specifies an environment for the experiment. This includes decisions such as thread parallelism, caching strategies (cold/hot caches), workload synthesis, and medium for the engine's storage. Next, the user sets the experiment parameters, such as metrics to collect, queries to run, dataset, target engines, etc. Finally, the user runs the experiment. When this finishes, the tool analyzes and plots the results, highlighting interesting insights based on a rule-based, statistical analysis of the results.

Figure 2[1] describes an experiment running the three queries on MonetDB and PostgreSQL and highlights some interesting points. First, PostgreSQL exploits multi-threaded execution, while MonetDB struggles here with multiple threads due to Python's Global Interpreter Lock (GIL). Second, the UDF in Q2 is implemented as a scalar in PostgreSQL that allows pipelined execution and as a table in MonetDB where it operates as a blocking operator (not an important problem in this simple query). Third, the impedance mismatch between the Python execution environment and the C++ database codebase imposes several overheads (e.g., data copies and conversions, context switches) that affect query performance. Q3 that has only one Python UDF performs better than Q1, as its SQL and C UDFs integrate more efficiently with the engine.

Scenario 2 - UDF and query optimization. The second scenario employs Q14 and Q16, two queries from query class QC3. Q14 identifies the most recent affiliation of the first author for publications funded by EU. This query presents optimization opportunities, such as pull-up optimization as an expensive table UDF (T-3) could be postponed until after a selective join. Q16 investigates how research projects affect the collaboration among scientists in terms of publications they co-authored. This query could benefit from operator fusion as several UDFs (and relational operators) could be fused into just two UDFs to eliminate overheads (e.g., context switches, data copies), reduce function calls, and potentially enable more performant compilation including the core functionality of the fused UDFs into the same hot-loop (loop fusion). Currently, not all engines support such optimizations, so in order to understand their potential benefit, UDFBench offers hand-made versions of such optimized queries to test (discussed shortly).

Figure 2[2] presents the execution of Q14 and Q16 on the four engines and highlights their differences. Multi-threaded execution benefits PostgreSQL and DuckDB, hurts MonetDB due to GIL, and does not affect SQLite that runs single-threaded. In Q14, the number of rows processed by the jsonparse UDF is the same on SQLite and PostgreSQL, indicating that these effectively apply UDF reordering (also verified in their plans). DuckDB does reordering but the rows processed differ slightly due to aggressive predicate push-down. In contrast, MonetDB does not apply UDF reordering and thus, has slower query runtime. Similar observations hold for Q16. However, although the pull up optimization was effective in Q14, in Q16 PostgreSQL suffers from this optimization, which results into high CPU time due to the excessive number of UDF calls. Finally, DuckDB

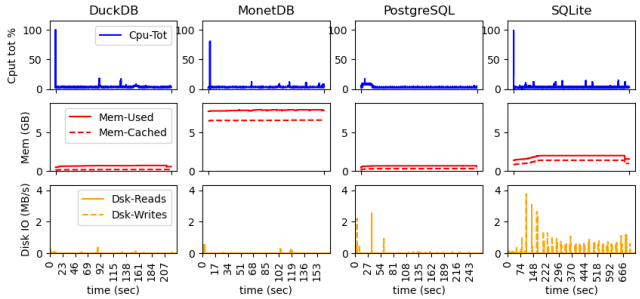


Figure 3: Resource utilization for query 16 (Q16)

processes similar number of rows as MonetDB, but with more UDF calls, indicating that it applies a batching mechanism. MonetDB functions as operator-at-a-time, hence, having the fewer UDF calls.

Scenario 3 - UDF optimization. The next scenario, shown in Figure 2[3,4], focuses on query Q16, which blends multiple UDFs with complex logic, and examines the effect of query and operator fusion optimizations. UDFBench provides Q16b, a manually optimized version of Q16 that fuses UDFs and relational operators into fewer, coarser-grained UDFs (Figure 2[3]) and thus, it reduces the context switches between the engine and UDF runtime. Q16b outperforms Q16 across all engines. Notably, the high variance of Q16 runtime across the four engines caused by their different execution models and function calls, has been optimized out in Q16b that involves only two (fused) UDFs implemented similarly across engines, reducing integration overhead. This experiment indicates a potential improvement for database engines; research prototypes have already made steps toward this direction [2, 3, 6, 12].

Scenario 4 - Architecture choice. Next, we examine how storage architecture affects query performance (Figure 2[5]), comparing Q14 on SSD vs. in-memory execution across the four engines. Column-stores (MonetDB, DuckDB) optimize I/O efficiently, so in-memory execution yields lesser gains. In contrast, row-stores (PostgreSQL, SQLite) are impacted more by disk I/O and hence, they benefit significantly from in-memory execution.

Scenario 5 - Resource utilization. The final scenario analyzes resource usage (cpu, memory, disk I/O) for Q16 (Figure 3). MonetDB is fastest but consumes memory aggressively. SQLite incurs frequent disk I/O, resulting into longer runtime (600+ sec). DuckDB and PostgreSQL leverage parallelism to improve performance.

Off-script presentation. For space considerations, we only described here a handful of queries and use cases supported in UDFBench. For off-script presentation, we would present additional queries and comparisons based on the audience interest to investigate additional features and limitations of the target systems, including: (a) stateful vs. stateless UDF execution, (b) deeper analysis of the impact of simple and complex data types in UDFs, as well as the size of data processed by UDFs, (c) just-in-time compilation of UDFs, (d) additional query optimization scenarios with complex UDFs (e.g., tf/idf score computation, classification/regression tasks, data/text mining algorithms, schema manipulation and cleansing transformations) and convoluted queries with several heavy operations (e.g., joins, aggregations, set operations), (e) investigate concurrent query execution, (f) explore the complexity of UDF queries through static code analysis, and so on.

5 Deployment

Our UDF queries are (■) or are not (□) supported by all engines. For example, SQLite and DuckDB do not support multilingual queries, such as Q3. The UDFBench UDFs are supported natively in PostgreSQL and MonetDB. However, table UDFs are not supported in SQLite and DuckDB, while the latter does not support aggregate UDFs either. To simulate table UDFs in DuckDB (■), we combine scalar UDFs with the SQL unnest operator. For aggregate UDFs, we implement the UDFs as scalar functions that receive the subquery string as a parameter. The UDF executes the subquery, returns a zero-copy dataframe, and computes the aggregation on it. In SQLite, we employ lazily evaluated, non-materialized, virtual tables that mimic the behavior of table UDFs [14] (■). Additionally, we employ APSW [1], a Python wrapper for SQLite to handle virtual tables and process the entire query set with minor query rewrite.

	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10	Q11	Q12	Q13	Q14	Q15	Q16	Q17	Q18	Q19	Q20	Q21
MonetDB	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■
PostgreSQL	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■
DuckDB	■	■	□	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■
SQLite	■	■	□	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■

Pluggability/Extensibility. To extend UDFBench to other engines we need to: (a) translate the load scripts for schemas and data, (b) translate the SQL queries, and (c) register the UDFs. To automate (a) and (b), we use a transpiler (SQLGlot [13]). A query rewrite might be needed to adapt to the diverse implementations of table UDF types across database engines (e.g., a table UDF of subtype 4 can be implemented in PostgreSQL as a scalar that returns a set, whereas in MonetDB, it must be implemented as a table UDF). The mechanism to register a UDF varies across database engines. For example, to register a Python UDF in MonetDB, we wrap it with a table function definition reading numpy arrays, whereas in PostgreSQL, it is defined as a scalar function returning a set of records.

Acknowledgments. This work was partially supported by the EU Horizon Europe programmes: DataGEMS (GA.101188416), CREXDATA (GA.101092749), and EBRAINS 2.0 (GA.101147319).

References

[1] APSW. 2024. Documentation, available at: https://rogerbinns.github.io/apsw.
[2] Konstantinos Chasialis et al. 2024. QFusor: A UDF Optimizer Plugin for SQL Databases. In *ICDE*.
[3] Andrew Crotty et al. 2015. Tupleware: "Big" Data, Big Analytics, Small Clusters. In *CIDR*.
[4] Yannis Foufoulas and Alkis Simitsis. 2023. Efficient Execution of User-Defined Functions in SQL Queries. *Proc. VLDB Endow.* 16, 12 (2023), 3874–3877.
[5] Yannis Foufoulas and Alkis Simitsis. 2023. User-Defined Functions in Modern Data Engines. In *ICDE*. IEEE, 3593–3598.
[6] Yannis E. Foufoulas, Alkis Simitsis, Eleftherios Stamatogiannakis, and Yannis E. Ioannidis. 2022. YeSQL: "You extend SQL" with Rich and Highly Performant User-Defined Functions in Relational Databases. *PVLDB* 15, 10 (2022), 2270–2283.
[7] Surabhi Gupta and Karthik Ramachandra. 2021. Procedural Extensions of SQL: Understanding their usage in the wild. *Proc. VLDB Endow.* 14, 8 (2021), 1378–1391.
[8] Viktor Leis et al. 2015. How Good Are Query Optimizers, Really? *Proc. VLDB Endow.* 9, 3 (2015), 204–215.
[9] Zhan Li, Olga Papaemmanouil, and Mitch Cherniack. 2016. OptMark: A Toolkit for Benchmarking Query Optimizers. In *CIKM*. ACM, 2155–2160.
[10] Patrick E. O'Neil, Elizabeth J. O'Neil, Xuedong Chen, and Stephen Revilak. 2009. The Star Schema Benchmark and Augmented Fact Table Indexing. In *TPCTC*.
[11] Fotis Psallidas et al. 2022. Data Science Through the Looking Glass: Analysis of Millions of GitHub Notebooks and ML.NET Pipelines. *SIGMOD Rec.* 51, 2 (2022).
[12] Leonhard F Spiegelberg, Rahul Yesantharao, Malt Schwarzkopf, and Tim Kraska. 2021. Tuplex: Data Science in Python at Native Code Speed. In *SIGMOD*.
[13] SQLGlot. 2024. SQLGlot, available at: https://github.com/tobymao/sqlglot.
[14] SQLite. 2022. SQLite docs, available at: https://www.sqlite.org/appfunc.html.
[15] Transaction Processing Performance Council. 2024. Active TPC Benchmarks, available at: https://www.tpc.org/information/benchmarks5.asp.