The UDFBench Benchmark for General-purpose UDF Queries

Yannis Foufoulas
Athena Research Center
Athens, Greece
johnfouf@athenarc.gr

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

Alkis Simitsis
Athena Research Center
Athens, Greece
alkis@athenarc.gr

ABSTRACT

User-defined functions (UDFs) extend the expressiveness of declarative SQL with functional capabilities, but also pose a core bottleneck in query processing due to the impedance mismatch between the UDF and SQL execution environments, and the limitations of the query optimizers to consistently produce good plans for UDF queries. Research and commercial approaches propose remedies for performant UDF query execution ranging from logical optimization and heuristics to physical optimization and compilation techniques. Each work however follows a different path to evaluate their proposed techniques. Despite the practical significance of optimizing UDF queries, UDFs have not been so far the focus of the database benchmarks. In this paper, we present UDFBench, a UDF-centric database benchmark based on real-world schema and data. We identify the core overheads in UDF query execution and design the UDFBench UDFs and queries to enable experimentation with these overheads, alone or in tandem with others. Finally, to showcase the portability and scope of UDFBench, we present an experimental analysis on five popular databases with different characteristics.

PVLDB Reference Format:
Yannis Foufoulas, Theoni Palaiologou, and Alkis Simitsis. The UDFBench Benchmark for General-purpose UDF Queries. PVLDB, 18(9): 2804-2817, 2025.
doi:10.14778/3746405.3746409

PVLDB Artifact Availability:
UDFBench code repo: https://github.com/athenarc/UDFBench.

1 INTRODUCTION

Relational database systems support functional extensions to SQL with user-defined functions (UDFs), which allow developers to implement complex logic and algorithms using a language of their choice. Most popular data engines support several types of UDFs (e.g., scalar, aggregate, table), but often UDF execution is neither optimized nor fully integrated with the underlying engine components such as the query optimizer and execution engine. And this comes at a cost, as the performance of executing queries with UDFs, or simply, *UDF queries*, inside a data engine is routinely subpar and creates significant bottlenecks largely due to the impedance mismatch between relational (SQL) evaluation and procedural (e.g., C/C++, Python, Java, Scala, R, JS) execution [19, 60, 61, 68].

Although external functions have been studied from the early 90's [e.g., 7, 28, 80], recently we experience an emerging interest in more advanced UDF functionality emanating from applications in data science and data analytics, including new and complex UDF types, such as analytic functions, ML models, ELT and continuous load functions [69, 71]. This trend has exacerbated the UDF query performance problem and has led to solutions employing low-level, physical optimization and compilation of UDF queries with an emphasis on UDFs coded in C/C++, Java, and Python [17, 18]. Python UDFs are particularly interesting as they (a) tend to be very popular among the growing communities of data science and data analytics [56], and (b) present intriguing and limiting performance challenges due to the conversions required between Python and C/C++, which is the implementation choice of most data engines.

As the performance deficiencies of UDF queries increasingly attract the spotlight, we need tools to benchmarking against best practices and results. Several database benchmarks have been proposed. The TPC benchmarks suite covers transactional, analytical, data integration, decision support, and other such workloads [76]. Other benchmarks include the Join Order Benchmark (JOB) [38], OptMark [39], Star Schema Benchmark (SSB) [49, 50], ETL-oriented benchmarks [70, 78], and frameworks such as the OLTPBench [12]. These focus on query processins challenges, such as join ordering, but they do not consider UDF queries. TPC-C and TPC-E contain a few store procedures, but as their focus is different, they do not capture the complexity and overheads UDFs pose to queries.

SQL-ProcBench is the only benchmark to date that focuses on procedural SQL workloads [24]. It employs an augmented TPC-DS schema with store procedures, user-defined functions, and triggers obtained from real-world applications. It is an starting point to study procedural SQL workloads, but it has several limitations. (a) The schema, object, and query definitions are expressed in three SQL dialects: PL/SQL [52], PL/pgSQL [54], and T-SQL [43], which makes it not fit to study the effect of UDFs coded in other popular languages such as C/C++ or Python. (b) SQL-ProcBench lacks support for popular UDF types. It comprises 24 scalar and 10 table UDFs, but not native aggregate UDFs. Its table UDFs are mainly blocking UDFs, corresponding to just 1 table UDF type (sub-type 5) out of the 8 types we use in our classification described later (see Figure 3). (c) Several of its UDFs are not used in the queries provided (e.g., genRandomChar, bestPromoStore). (d) And those they do are invoked by simple SQL queries ('select udf from dual'), while its few more complex queries employ just a single UDF. But to understand and measure the implications of UDF query execution, we need to explore more complex combinations of UDFs and relational operators in the same query and to systematically organize UDFs based on their characteristics. (e) Finally, although SQL-ProcBench can evaluate specific techniques such as inlining, it is not designed to test other critical dimensions as we discuss in Section 2.

In this paper, we present the UDFBench, a UDF-centric database benchmark that focuses on UDFs and their implications to SQL query performance. UDFBench is based on an extensive analysis

Proceedings of the VLDB Endowment, Vol. 18, No. 9 ISSN 2150-8097.
doi:10.14778/3746405.3746409

of research and commercial solutions related to UDF query processing [17, 18] to identify (a) the core challenges in UDF query optimization and execution at both (physio-)logical and physical levels, (b) the design complexity due to the large variety of UDF and data types involved, (c) the complexity of dealing with multiple programming languages, (d) the multiplicity of execution environments and how these affect UDF execution, and (e) the expresiveness expectations developers have for procedural programming in data engines. We follow a choke point analysis to identify the core bottlenecks in UDF query execution and designed the UDFBench to enable studying and measuring each of these overheads, alone or in tandem with others. UDFBench utilizes a database schema, data, and queries inspired from a real-world application. To demonstrate the portability of the benchmark, we present experimental results on five data engines with different characteristics.

Our contributions. UDFBench is designed to study the limitations and potential opportunities in current approaches, hoping that it could influence future work toward more performant UDF execution. Our contributions are summarized as follows:

(a) We describe the challenges and complexity of UDF execution, and the core factors of UDF performance deficiencies.

(b) We present UDFBench, a benchmark based on a real-world application and data that treats UDFs as a first-class citizen and is specifically tailored for UDF queries in SQL databases.

(c) We showcase the portability of UDFBench with an experimental evaluation on five data engines with different architectures.

(d) We identify performance deficiencies and design limitations that indicate potentially fruitful, future research directions.

(e) The implementation of UDFBench follows a modular design that could be straightforwardly extended to additional queries, programming languages, and data engines. To that end, we open-source all the bits of UDFBench, schema, data, and UDF queries [77].

2 DESIGN PRINCIPLES

Our design process relies on choke points analysis [13] to identify critical factors in UDF query processing as listed next.

UDF programming language. Most SQL engines support UDFs in various languages, with PLSQL, C/C++, Python, and Java being among the most popular choices. Table 1 presents a listing of UDF language support in popular SQL engines. We used the first five engines as representative engines in our evaluation presented in Section 4. Furthermore, a SQL query may contain UDFs coded in more than one language, which complicates further its performance analysis. For example, the query: '`select median(jsoncount(citations)) from result`', may involve `median` as a C UDF and `jsoncount` as a Python UDF. Although early research efforts handle such queries as poly-glot or federated queries [22, 30, 31, 72], SQL engines could be a performant one-stop shop for multi-language UDF queries.

UDF types. Modern SQL engines support a variety of UDF types, including scalar, aggregate, and table UDFs. Scalar and aggregate UDFs process one tuple or one group at a time, respectively, and return a single value. Table UDFs process a subquery or scalar parameters at a time and return a table. Other UDF types include Lambda expressions (especially, for scalar UDFs), window functions, selection functions, and so on. However, as these are still not widely supported, we do not cover them in the first version of UDFBench.

SQL engine	UDF languages
DuckDB	Python/Pandas, C
MonetDB	C, Python, R
PostgreSQL	pgSQL, Tcl, Perl, C, Python
SQLite	C, Python (with APSW)
Spark	Scala, Java, Python (PySpark)
Databricks	Hive, Python/Pandas, Scala
DB2	C++, Lua, Python, R
Greenplum	C, Python, pgSQL, Tcl
MariaDB	C/C++, Rust
MongoDB	JS
Oracle	PL/SQL, C, Java, Python (OML4Py)
Redshift	SQL UDF, Python, Lambda (Java, Go, C#, Ruby, etc.)
Snowflake	Java, JS, Python, Scala, SQL UDFs
SQL Server	Transact-SQL (TSQL)
Vertica	C++, Java, Python, R

Table 1: Example UDF language support in SQL engines

Data types. The data types used in the UDF input and output schemata may be simple (int, float, string, null) or complex (json, dicts, lists). Simple data types are typically aligned with the underlying engine. For example, a Python UDF running on a C++ based engine should deal with data conversion and data copy issues. Numeric values are seamlessly processed between the Python and C++ runtimes via cdata objects. String values are transformed into a format processable by either runtime via pointers to cdata objects with or without (interned) data copies, with methods such as a compiler friendly `ffi.string`, `ffi.buffer` (memoryview without string copy), or `direct pass` the pointer to the C string enabling low level optimizations in a C manner. Complex data types introduce additional complexity, especially if the engine handles these types via JSON strings. Then, the UDF developer should (de)serialize the native structure to JSON, which adds an overhead to data conversions and switches. For example: `Python struct` → `Python JSON string` → `C/C++ string`. UDFBench includes UDFs with both simple and complex data types to allow measuring such overheads.

Example. The query '`select extractmonth(date) from artifacts`' employs a scalar UDF `extractmonth` with three semantically equivalent variants but different return data type: integer (`1-12`), string (`Jan-Dec`), and JSON (`{'month':'Jan'}`). Figure 1(left) shows the data conversion and copy overheads in (a) the UDF body and (b) the (de)serialization between the UDF and the engine, here MonetDB.

Execution mode. UDF execution depends on the characteristics of the underlying engine, such as thread/process parallelism, in/out process execution, and tuple/operator at a time execution. Multi-threaded query execution improves query performance and scalability. However, parallelism in UDF execution is limited by several factors [20]. For example, the performance of Python UDFs run in parallel is limited by the Python Global Interpreter Lock (GIL) [58], a mutex that allows only one thread to control the Python interpreter hence forcing the Python program to run single-threaded. To remediate this, multi-process execution could be employed instead. Additionally, the UDF may be executed in the same or separate process with the engine. The former mode is performant as there is little handshake between the UDF and the engine's runtime, but it could become a potential vulnerability in a malformed execution. The out-process execution adds a safety net for the engine at the cost of potentially increased communication overheads at the UDF execution. Performance trade-offs also exist when the UDF runs per tuple resulting into numerous functions calls, versus running in vectorized mode, which may break the execution pipeline and introduce intermediate results. The query and UDF design in our

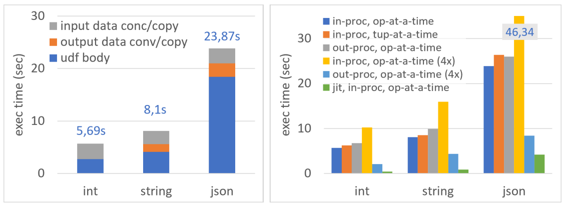


Figure 1: Query overheads related to data types (left) and execution modes (right) on the large dataset

benchmark creates opportunities to measure such execution modes.

Example. Figure 1(right) shows a comparison of out-of-the-box query execution on MonetDB (i.e., in-process (`in-proc`) and operator-at-a-time (`op-at-a-time`)) with other combinations of hand-crafted execution modes including out-process (`out-proc`), tuple-at-a-time (`tup-at-a-time`), multi-threaded (`in-proc 4x`, with 4 parallel threads), multi-processed (`out-proc 4x`, with 4 parallel processes), and JIT-compiled (`jit`). Several observations can be made: (a) the difference between `op-at-a-time` and `tup-at-a-time` (the first two bars) is attributed to the excess function calls due to per tuple function invocation, (b) `out-proc` is slower than `in-proc`, (c) thread-level parallelism is slower than process-level parallelism due to GIL, (d) a combination of in-process, operator-at-a-time, process parallelism and JIT-compiled UDFs is a good choice for this particular engine. We investigate such design options in more detail in Section 4.

Query optimization. Query optimizers generally treat UDFs as a black-box. Early approaches identified that computationally expensive and complex functions become dominant cost factors in query optimization, whilst conventional heuristics often do not work [7–9, 27, 28, 80]. Later approaches explored factorization of shared subexpressions (e.g., UDF calls) for finding optimal orderings of predicates and functions, which is an NP-hard problem [46, 47]. Techniques such as introspection [5, 29] and code analysis enable logical optimization, such as operator re-ordering or UDF push-down, a problem recently revisited using a search-verification approach [81]. Adaptive query execution (ADE) improves UDF query optimization, as it deals with the varying UDF cost in a query's lifespan [11, 23, 33, 40, 79]. For example, interleaved execution depends on live query statistics for multi-statement table-valued functions, a specific type of table UDFs with T-SQL statements [63]. UDFBench is a testbed for such techniques, as its queries blend relational operators and UDFs in a variety of combinations offering query optimization opportunities of an increasing complexity, ranging from relatively easy to capture to more challenging cases.

UDF optimization. UDF query execution can benefit from techniques such as parallelization, vectorization, function inlining/outlining, loop fusion, and just-in-time (JIT) compilation [2, 10, 14–16, 19, 20, 25, 32, 34, 53, 59–61, 65–68, 73]. Approaches to physical optimization of UDF queries differ to each other in the method of UDF integration with the data engine. Frequent choices include UDF translation to either SQL or to an internal representation (IR), and engine-level UDF compilation and integration [17, 18]. UDFBench is designed to enable testing the benefits, implications, and limitations of such low-level techniques. For example, it comprises queries with varying UDF chaining and nesting levels, as follows: `select pubmedid, jpack(frequentterms(stem(filterstopwords( keywords(abstract))),10)) as pmcterms from file('pubmed.json')`. This query could presumably benefit from loop fusion [3, 6, 34], function inlining/outlining [2, 16, 20, 60], vectorization [20, 35, 36, 59], and JIT-compilation [21, 37, 45, 62, 64, 73], should the underlying execution engine supports any of these techniques.

Expressiveness. Procedural programming allows dynamic/static typed functions, parametric polymorphism, and stateful execution. These features are sparsely supported for UDFs in SQL engines. For example, a *dynamically typed* UDF such as `add(a,b)` could be instantiated with various types at query time: `select cast(add(1,2) as int)` or `select cast(add('hello','world') as string)`. Similarly, a *polymorphic* function such as `file` in '`select authors from file('pubmed.json')`', should allow to dynamically determine the schema of `authors` based on the format of the input; e.g., the `authors` could be either `{'name':'Peter'}` or `{'name':'Peter','citedby':100}`. Finally, a *stateful* function should retain the state in between query runs, e.g., after a variable instantiation with '`select var('a','HELLO')`', the query '`select lower(var('a'))`' should return `hello`.

We have designed UDFBench to enable investigating and measuring such factors across data engines and research solutions.

3 THE UDFBENCH BENCHMARK

UDFBench comprises a schema populated with real-world data, UDFs of varying complexity, and queries containing a blend of relational and UDF operators. A complete listing of UDFBench queries and UDFs may be found in our code repository [77].

3.1 Schema

3.1.1 Schema origin. The UDFBench schema hails from a real-world application, OpenAIRE (openaire.eu), which uses 150+ UDFs to perform information extraction, text mining and analytics over Open Access publications. To date, it has harvested over 130M publications, 2M datasets, 85K software artifacts, and 1.5M projects. UDFBench schema is a curated subset of OpenAIRE's schema [41].

3.1.2 Data origin. UDFBench comprises real-world data at three scale factors: small (13M rows), medium (60M rows), and large (120M rows). These are adequate to show the implications of UDFs in UDF queries. Larger datasets would mingle the UDF overheads (i.e., the focus of UDFBench) with other factors (e.g., increased I/O or disk spills). Based on demand, we could offer a generator of synthetic data from the real dataset at arbitrary sizes [4].

3.1.3 Schema description. Figure 2 depicts the ER-diagram of the UDFBench schema, which comprises 10 tables as shown in Table 2. It has 2 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. Note that the schema contains denormalized features such as nested attributes (JSON list, dict) and the queries presented shortly connect to external denormalized files, hence, enabling testing complex UDF functionality beyond relational. The schema includes fields of various data types and sizes, and allows building queries with expensive relational operators such as aggregation, grouping, and join paths of moderate length (up to 5). We find this fit for benchmarking UDF queries. For studying problems such as join ordering in many-join queries, one could use specialized benchmarks such as JOB [38] or SSB [49, 50].

id	table name	table description	#fields	row counts			disk size (MB)		
				small	medium	large	small	medium	large
T1	artifacts	artifacts (e.g., publications, datasets)	17	376,152	1,880,762	3,761,525	112	560	1,119
T2	artifact_abstracts	abstracts of the artifacts	2	137,454	686,165	1,372,429	98	489	1,345
T3	artifact_authorlists	author(s) of each artifact (in JSON)	2	127,269	635,753	1,271,629	25	122	243
T4	artifact_authors	author metadata	7	1,022,184	4,956,523	9,931,641	305	1,469	2,946
T5	artifact_charges	artifact processing charges	3	17,057	85,287	170,574	1	5	10
T6	artifact_citations	citation count per artifact	3	15,218	76,090	156,749	7	25	38
T7	projects	project metadata	24	469,604	1,653,651	3,307,303	132	447	890
T8	projects_artifacts	links between projects and artifacts	3	628,274	3,144,975	6,578,760	71	354	741
T9	project_artifactcount	artifacts per project	5	469,604	1,653,651	3,307,303	29	101	201
T10	views_stats	statistics about artifact views	5	9,686,539	44,960,583	89,921,167	1,153	5,353	10,619
		total:	71	12,949,355	59,733,440	119,779,080	1,933	8,916	18,152

Table 2: UDFBench tables

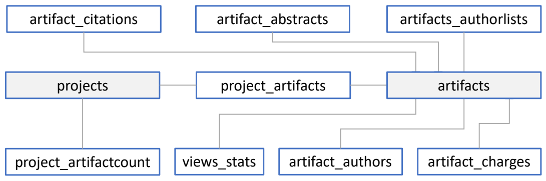


Figure 2: UDFBench: Schema

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 3: UDF type (type) and sub-type (sub) classification

3.2 User-defined functions

UDFBench contains 42 UDFs inspired by production code in OpenAIRE [51], which are carefully crafted to enable experimentation with the factors and challenges described in Section 2.

3.2.1 UDF classification. We consider the most popular UDF types: scalar, aggregate, and table UDFs. Their input and output schemata are critical in how a UDF operates in a query and how it interacts with its producer and consumer operators. Based on this, we classify these three types into 10 sub-types (sub), as shown in Table 3. Scalar and aggregate UDFs have always the same mapping from their input to their output. Table UDFs are more complicated; they have 8 mappings of their input to their output schemata, e.g., process a table and return a tuple, or process a table and return a table.

Each sub-type in Table 3 enables different optimization opportunities. Table UDFs that process one tuple at a time (T-3, T-4) are parallelizable via data partitioning, while those processing the entire table at once are blocking. For example, the `strsplitv` UDF splits a string's token into separate rows (one tuple at a time). Each input text can be processed independently, still some engines such as MonetDB treat `strsplitv` as a black-box table function. Table UDF sub-types T-6 and T-7 process one group at a time. Unlike aggregate UDFs, they return multiple values but enable similar optimizations. For example, the `top` UDF processes a group at a time, returning the max N values as separate rows.

3.2.2 UDF design and characteristics. Table 4 presents the UDFBench UDFs along with their types, sub-types, and complexity. The list contains 24 scalar, 4 aggregate, and 14 table UDFs of various sub-types. Table 5(left) shows statistics including the cardinality of UDFs that have a pipeline or blocking execution, can be parallelized, are stateful, (may) have side-effects, and are supported in more than one programming language. Table 5(right) shows statistics related to the data types of the UDF input and output schemata.

3.2.3 UDF description (abridged). UDF implementation is engine specific. UDFBench exploits each data engine's UDF capabilities and adapts to differences in type definitions and execution models across engines. For example, `extractfromdate` (table, T-3) may be implemented either as a blocking table UDF or as a parallelizable scalar UDF. Next, we describe example UDF design choices for polymorphic/dynamic UDFs, stateful UDFs, and parallelism in UDFs.

Dynamic and polymorphic UDFs. Polymorphic UDFs are implemented once and can be reused with varying input/output types and different arguments in each invocation. Some engines, such as PostgreSQL, support polymorphic UDFs and permit data type specification at query time, hence handling multiple types dynamically. Systems lacking built-in support for UDF polymorphism, require the UDF developer define a UDF multiple times -one for each type variation- to express queries in SQL-standard syntax. The following example lists an implementation of the table UDF `file` (U40) in (a) MonetDB and (b) PostgreSQL, and two example SQL queries.

```
(U40-a):  create or replace function file_v1(fpath string, ftype string)
          returns table (col1 string, col2 string, col3 string, col4 string)
          language python {
             -- udf body that returns a specific output schema
          };
          create or replace function file_v2(fpath string, ftype string)
          returns table (col1 string, col2 string, col3 string)
          language python {
             -- udf body that returns a different output schema
          };
          [SQL query]: select * from file_v1('arxiv.xml','xml');
(U40-b):  create or replace function file(fpath text, ftype text)
          returns setof record as $$
            -- udf body that returns dynamic schemas according to inputs
          $$ language plpython3u;
          [SQL query]: select * from file("arxiv.xml","xml")
                         f(doi text, amount float, totalpubs int, sdate text);
```

Stateful UDFs. Implementing stateful UDFs depends on how an engine handles persistent resources. Consider the `keywords` UDF (U20), which tokenizes text input based on a regular expression. (a) SQLite allows reusing a pre-compiled regex pattern across calls by defining it at the module level, so it maintains state and avoids repeated compilation overhead. (b) PostgreSQL PL/Python does not support persistent state within UDFs across calls, so the regex must be recompiled each time the UDF runs, leading to higher overhead.

id	name	type	cost
U1	Addnoise	S-1	$O(n)$
U2	Clean	S-1	$O(n)$
U3	Cleandate	S-1	$O(n)$
U4	Converttoeuro	S-1	$O(n)$
U5	Extractclass	S-1	$O(n)$
U6	Extractcode	S-1	$O(n)$
U7	Extractday	S-1	$O(n)$
U8	Extractfunder	S-1	$O(n)$
U9	Extractid	S-1	$O(n)$
U10	Extractmonth	S-1	$O(n)$
U11	Extractprojectid	S-1	$O(n)$
U12	Extractyear	S-1	$O(n)$
U13	Filterstopwords	S-1	$O(n)$
U14	Frequentterms	S-1	$O(n)$
U15	Jaccard	S-1	$O(n * k)$
U16	Jpack	S-1	$O(n * k)$
U17	Jsoncount	S-1	$O(n)$
U18	Jsort	S-1	$O(n)$
U19	Jsortvalues	S-1	$O(n)$
U20	Keywords	S-1	$O(n)$
U21	Log10	S-1	$O(n)$
U22	Lower	S-1	$O(n)$
U23	Removeshortterms	S-1	$O(n)$
U24	Stem	S-1	$O(n)$
U25	Avg	A-2	$O(n)$
U26	Count	A-2	$O(n)$
U27	Max	A-2	$O(n)$
U28	Median	A-2	$O(n * logn)$
U29	Extractfromdate	T-3	$O(n)$
U30	Jsonparse	T-3	$O(n * k)$
U31	Combinations	T-4	$O(n * k^2)$
U32	Extractkeys	T-4	$O(n * k)$
U33	Strsplitv	T-4	$O(n * k)$
U34	Jgroupordered	T-5	$O(n)$
U35	Kmeans (iterative)	T-5	$O(n^2)$
U36	Kmeans (recursive)	T-5	$O(n^2)$
U37	Xmlparse	T-5	$O(n)$
U38	Pivot	T-6	$O(n * k)$
U39	Top	T-7	$O(n * logk)$
U40	File	T-8	$O(n)$
U41	Output	T-9	$O(n)$
U42	Getstats	T-10	$O(n * logn)$

Table 4: UDFBench UDFs: type (scalar-S, aggregate-A, table-T), sub-type (1-10), and cost

type	count
scalar	24
aggregate	4
table	14
pipeline	36
blocking	6
parallelizable	32
stateful	14
side-effect	1
multi-lang	3

data type	in	out
numeric	8	1
int	0	6
float	0	6
string	22	16
rset	0	1
json	4	1
xml	1	0
bool	0	1
polymorphic	7	10

Table 5: UDF characteristics

```
(U20-a):  text tokens = re.compile(r'([\d.]+\b|\w+)', re.UNICODE)
          ┌ keywords udf ──────────────────┐
          │ res = text_tokens.findall(input) │
          └──────────────────────────────────┘

(U20-b):  ┌ keywords udf ─────────────────────────────────────────┐
          │ text_tokens = re.compile(r'([\d.]+\b|\w+)', re.UNICODE) │
          │ res = text_tokens.findall(input)                        │
          └─────────────────────────────────────────────────────────┘
```

Parallel UDFs. Scalar UDFs, which operate on a row-to-row basis, are typically straightforward to parallelize. Aggregate UDFs can be parallelized, especially, if implemented with an `init-combine-final` model to merge partitioned results. Table UDFs are generally more complex to parallelize. UDF parallelization can also be directed by the developer. For example, in PostgreSQL, tagging a UDF as `PARALLEL SAFE` enables parallel execution, assuming that its operations are safe for concurrent processing. Hence, with `PARALLEL SAFE`, PostgreSQL executes `extractfromdate` across parallel workers.

```
(U29):  create type _extractfromdate as ( extractyear integer,
            extractmonth integer, extractday integer );
        create or replace function extractfromdate(arg text)
        returns _extractfromdate as $$
            -- udf body that converts date to an _extractfromdate type
        $$ language plpython3u immutable strict parallel safe;
```

3.3 Queries

3.3.1 Query classification. Following the principles described in Section 2, we categorize the benchmark queries into four query classes.

(QC1) Queries with UDFs and a few relational operators (emphasis on UDFs): useful for evaluating UDF related overheads.
(QC2) Queries with a blend of UDFs and expensive relational operators (e.g., join, group) or nesting: useful for evaluating the interaction of UDFs with heavy 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).

3.3.2 Query design and characteristics. UDFBench contains 21 queries inspired by real-world use cases, fine-tuned to enable investigating the challenges described in Section 2. Table 6 presents their characteristics: (a) query class (QC1 to QC4), (b) #UDFs per UDF type, (c) data type of the query input/output either simple (text, numeric, etc.) or complex (JSON, list, dict, etc.), (d) may process null values, (e) contains a UDF chain (nested UDFs), its UDF and query nesting depth, (f) could benefit from optimizations such as JIT compilation, fusion, vectorization, parallelization, (g) could operate in-process, (h) contains dynamic, polymorphic, stateful UDFs, (i) number of expensive relational operators, etc.

3.3.3 Query description (abridged). Next, we describe representative UDFBench queries per query class.

QC1 - Simple UDF queries. These queries are useful to analyze the impact of UDF specific characteristics, such as UDF type, language, low-level overheads and potential UDF optimizations.

(a) Extract the year, month, and day from date. (Q1): use 3 scalar UDFs that input the date as a string and produce three integer values. (Q2): use a table UDF that inputs one string column (date) and produces three integer columns. (Q3): same as Q1, but the 3 UDFs are implemented in different languages (Python, C, SQL, Java, Scala).

```
(Q1):  select id, extractyear(date), extractmonth(date), extract-
       day(date) from artifacts;
(Q2):  select id, extractfromdate(date) from artifacts;
(Q3):  select id, extractyear(date), extractmonth_c(date), extract-
       day_sql(date) from artifacts;
```

(b) Similarly, we include UDF queries designed to test simple data types (Q4, Q5) e.g., numeric, float, and complex data types (Q6-Q9) e.g., JSON list, XML, with various UDF types.

QC2 - Mix of UDFs and relational operators. These queries are useful for exploring potential query optimization opportunities, pipeline or blocking processing, context switches and data copies and conversions between UDF and relational operators.

(a) Clustering of artifact types based on the funded amounts of their linked projects. Alternative implementations: (Q10): use iterative k-means clustering. (Q11): use recursive k-means clustering.

```
(Q10): select * from kmeans(
         'select id, type, sum(famt) as sfamt from (
         select a.id as id, a.type, converttoeuro( p.fundedamount, p.currency) as famt
         from artifacts a, projects p, projects_artifacts pa
         where a.id = pa.artifact_id and pa.project_id = p.id and fundedamount>0.0 )
         group by id, type', 'type', 'sfamt', 'id', 5);
```

Some engines (e.g., PostgreSQL, MonetDB) execute Q10 as two queries, hence enabling query optimization that could otherwise be blocked by `kmeans` UDF: (a) a query with the k-means UDF and (b) a query that passes as an argument in the k-means UDF. In the latter query, the `converttoeuro` UDF could be pushed before the join.

(b) Combine UDFs with join and group-by. (Q12): Show the 10 most viewed publications in the last year and add Gaussian noise to hide the specific number of views. This is a top-k query that applies a UDF per artifact. Q12 explores low-level UDF optimizations (e.g., loop fusion) in synergy with SQL expressions, and overheads involved for executing two UDFs with grouping and sorting.

```
(Q12): select artifact_id, addnoise(count(*)::int) as views
         from views_stats where cleandate(date)::timestamp with
         time zone >= now() - interval '12 months'
         group by artifact_id order by views desc limit 10;
```

query class	QC1 (simple UDF queries)									QC2 (UDFs + expensive RelOps)				QC3 (complex UDF queries)						QC4 (UDFs + DML)	
query id	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
#scalar	3		3				2	2	1	1	1	2	1	4	3	10	5	11		1	1
#aggr				2			2	2	1	1	1	1		1		3	1	1			
#table		1			1	3	1		1*	1	1		2		2	1*	2	2	1		2
int	x	x	x	x	x		x	x	x			x				x	x	x	x		
string	x	x	x			x	x	x	x	x	x		x	x	x	x	x	x	x	x	x
float				x	x		x	x		x	x	x					x	x			
null	x	x	x	x	x		x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
complex						x	x	x	x				x	x	x	x		x			x
parallel	x		x	x		x	x	x		x	x			x	x	x	x	x			
in-process	x	x	x	x	x	x	x	x		x	x	x	x	x	x	x	x	x	x	x	x
udf chain							x	x	x	x	x		x		x	x	x	x	x		x
jit	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
fusion/loop fusion	x		x	x		x	x	x	x	x	x		x	x	x	x	x	x			x
vectorization	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
dynamic/polymorphic					x	x	x	x	x						x	x		x	x		
stateful	x		x								x				x	x	x	x		x	
#rel-ops						1				5	5	5		15	6	12	4	1	2		
cost model										x	x	x		x	x	x	x	x			
nest udf depth	1	1	1	1	1	2	3	2	3	3	3	1	3	2	3	5	6	7	1	1	3
nest qry depth	1	1	1	1	1	2	2	1	3	3	3	2	4	3	3	4	4	3	2	1	4

Table 6: UDFBench: List of queries with their characteristics

(c) Table UDFs with nested subqueries. (Q13): Parse an external JSON file and extract information to link publications with projects.

QC3 - UDFs and complex relational logic. These queries are useful for exploring advanced query optimization and low-level, UDF optimization opportunities, involving expensive operators and deep nested UDFs and subqueries with expensive context switches, data copies and conversions. As these may be not handled by existing query optimizers, we also offer hand-crafted optimized alternative query implementations to highlight the potential.

(a) (Q14): Find the most recent affiliation of the first author for publications funded by EU. This query could benefit from a UDF pull-up optimization [27, 28] as the expensive `jsonparse` UDF could be postponed until after the selective join (Q14').

```
(Q14):  with aa (ath,aff,..) as (
            select jsonparse(authorid,..), jsonparse(affiliation,.. ), [..]
            from artifacts_authors where [..4 filters..])
        select aa.ath, aa.aff
        from aa, artifacts a, projects_artifacts pr, projects p
        where [..3 filters / 1 subquery with join/group..] and
            p⋈pr and pr⋈a and aa⋈a;
(Q14'): select jsonparse(authorid,..), jsonparse(affiliation,..)
        from artifact_authors, artifacts, projects_artifacts, projects
        where [..7 filters / 1 subquery with join/group..] and
            p⋈pr and pr⋈a and aa⋈a;
```

(b) (Q16): Investigate how research projects affect the collaboration among scientists in terms of publications they co-authored.

```
(Q16): with pairs as (
     ⎧ select [..], extractid(), extractfunder(), extractclass(),
   A ⎨ combinations(jsort(jsortvalues(removeshortterms(lower([..])))))
     ⎩ from artifacts )
       select funder, class, projectid,
     ⎧ sum(case: cleandate() between projectstart and projectend [..])
     ⎪   as authors_during,
   C ⎨ sum(case: cleandate() < projectstart [..]) as authors_before,
     ⎩ sum(case: cleandate() > projectend [..]) as authors_after
       from ( [..two nested subqueries..] ) as  projectpairs ⋈ pairs )
       group by funder, class, projectid;       └──────┬──────────┘
                                                       B
```

This query presents several optimization opportunities. The UDFs in the A and/or C portions could be fused, respectively, into two single UDFs. This would eliminate overheads such as context switches and data copies, reduce function calls, potentially enabling more performant compilation including the core functionality of the fused UDFs into the same hot-loop (loop fusion). Then, the query plan abstractly becomes $A \longrightarrow B \longrightarrow C$. This presents optimization opportunities, such as reordering of UDFs and relational operators, predicate push-down (e.g., search only for European funders) [81] or predicate pull-up (e.g., postpone time-consuming or resource-consuming UDFs after a selective join operation) [28].

Similar opportunities and challenges could be explored with the other queries of this category that implement functionality such as:

(c) (Q15): Search crossref (an XML file) to find publication-project pairs that do not exist in the local dataset.

(d) (Q17): TF/IDF computation for artifact abstracts.

(e) (Q18): Identify the top-5 most similar documents for a given document. The query applies several preprocessing steps and computes the Jaccard similarity on document abstracts from 2 different sources (csv and JSON files) for each pair of documents.

(f) (Q19): Employ a pivot operator to count the number of artifacts per artifact_type and per project.

QC4 - UDFs and DML operations. These queries investigate the impact of UDF execution on typical DML queries (insert, update).
(a) (Q20): Update dates in the `artifacts` table after a cleaning task.
(b) (Q21): Extract links between publications and projects from an external JSON file and insert them to the `projects_artifacts` table.

```
(Q20): update artifacts set date = cleandate(date);
(Q21): insert into projects_artifacts
         select publicationdoi, crossref.pid from (
            select publicationdoi, extractprojectid(fundinginfo) as pid
            from ( select * from jsonparse(..) as t ) as crossref );
```

3.4 Parameters and Metrics

We consider the following metrics associated with UDFBench.

3.4.1 Performance. To measure the overheads of UDF execution, we use: (a) execution time: it measures the time from the moment a query is issued until the result is retrieved; (b) resource utilization: the average cpu/memory utilization for a query execution; (c) process/read size: the size of data (MBs) processed/read by a query.

3.4.2 Software metrics. To measure the complexity and expressiveness of the proposed UDFs, we employ software engineering techniques, such as the Halstead metrics, McCabe's cyclomatic complexity (CC), the maintainability index (MI), and raw metrics, such as total (LOC), logical (LLOC), and source (SLOC) lines of code.

The Halstead metrics assess a program's complexity and statically analyze the source code to compute the number of distinct (η_1 and η_2) and total (N_1 and N_2) operators and operands [26]. From these, we compute the measures: (a) program vocabulary: $\eta = \eta_1 + \eta_2$, (b) program length: $N = N_1 + N_2$, (c) calculated program length: $\hat{N} = \eta_1 \log_2 \eta_1 + \eta_2 \log_2 \eta_2$, (d) volume: $V = N \times \log_2 \eta$, (e) difficulty to write/understand a program: $D = \frac{\eta_1}{2} \times \frac{N_2}{\eta_2}$, (f) effort to code: $E = D \times V$, (g) estimated time to code: $T = {}^{E}/_{18}$ seconds, and (h) estimated number of bugs in the code: $B = {}^{V}/_{3000}$. These metrics help understand the diversity and size of the code, aiding in evaluating its complexity and potential maintainability, and hence they provide valuable insights into the code's structure and complexity.

The cyclomatic complexity considers conditional logic in blocks measuring the linearly independent paths in the code [42]. Programs with lower cyclomatic complexity are easier to understand and less risky to modify. The maintainability index shows the relative maintenance effort for blocks of code and is computed as a factored formula of Halstead's volume, the cyclomatic complexity, the lines of code (LOC), and the percentage of comments [48, 75]. We use these metrics as a 'rule of thumb' to compare the *relative* trends of the UDF implementations across different systems.

4 PERFORMANCE EXPERIMENTS

We present an experimental evaluation on five SQL engines to showcase how UDFBench could be used to evaluate various features and techniques related to UDF implementation and architecture, and we fine-tune the engines for this purpose. However, this analysis should not be used as an engines comparison, this is not our target.

4.1 Experimental setup

Single-node experiments ran on a server with 2x Xeon E5-2630V4 (40pt), 144GB DDR4, and Ubuntu 22.04, which was otherwise unloaded. The systems/tools used include: PostgreSQL (17), MonetDB 5 server (`v11.50.0`), DuckDB (`1.0.0`), SQLite3 (`3.37.2`), Spark/PySpark (`3.5.5`), Python (`3.10.12`), gcc (`11.4.0`), PyPy (`7.3.12`), CFFI (`1.15.1`), APSW (`3.40.0.0`), NumPy (`1.26.4`), PyArrow (`18.0.0`), Pandas (`2.2.0`), Scikit-learn (`1.4.0`), NLTK (`3.8.1`), Radon `6.0.1` [57].

4.2 Methodology

Engines. We deployed UDFBench to DuckDB, MonetDB, PostgreSQL, SQLite, and Spark, which represent various architectures: embedded vs. server-based, memory-based vs. disk-based, row-store vs. column-store, and single-node vs. distributed.

Engine compatibility. The UDFBench queries capture aspects of UDF design options that may (🟩) or may not (□) be currently supported by all engines. Also, the queries may slightly differ across the engines due to differences in SQL dialects and the UDF types each engine supports. Next, we list a query compatibility matrix.

	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	🟩	■	□	🟩	■	■	■	🟩	■	■	■	🟩	■	🟩	■	■	■	■	■	🟩	■
Spark	🟩	🟩	🟩	🟩	□	□	🟩	🟩	🟩	□	□	🟩	🟩	🟩	🟩	🟩	□	🟩	□	□	□

In general, PostgreSQL and MonetDB support UDFBench UDFs natively with minor adjustment to align with how they employ UDF types. DuckDB, SQLite, and Spark do not currently support all UDF types. Next, we describe how we evaluated them with UDFBench.

DuckDB supports scalar UDFs, but not aggregate or table UDFs (except UDFs of sub-type 3). Instead of reporting 'n/a', we provide an alternative implementation (■). To simulate table UDFs in DuckDB, we combine scalar UDFs with the SQL unnest operator. This applies to queries Q6, Q7, Q9, Q10, Q11, Q13, Q15, Q16, Q17, Q18, Q19, and Q21. Similarly, in queries involving aggregate UDFs without a group-by, we implement the UDFs as scalar functions that receive the subquery string as a parameter. The UDF executes the subquery and returns a Pandas dataframe without data copy. The aggregation is then computed directly on the dataframe, and the final result is returned. This applies to queries Q4, Q5, Q7, Q8, and Q9. Queries involving aggregations with group-by are implemented as a combination of the above two techniques (e.g., Q14).

SQLite supports scalar and aggregate UDFs but not table UDFs. To simulate table UDFs (■), we use lazily evaluated, non-materialized, virtual tables that mimic the behavior of table-returning UDFs [74]. We also employ APSW [1], a Python wrapper for SQLite, to rewrite the query to create a lazy virtual table and then use it in the query.

Spark performance and scalability relies on aggressive partition parallelism. Hence, Spark applies the same principle to UDF execution as well. Spark UDFs run as out-process. Scalar UDFs run tuple-at-a-time, whereas aggregate UDFs are vectorized via Pandas. Spark requires that table UDFs (new in Spark 3.5) can be safely parallelized, which is true for those UDF sub-types that have tuple or value input type and process tuples independently (e.g., sub-types 3, 4, 8 in Table 3). But UDFs with other input types (e.g., tables or groups) are not always parallelizable. The blocking UDF sub-types 5, 9, and 10 process the full table at once and are not fit for partitioned parallelism. UDF sub-types 6 and 7 operate on grouped data and without careful handling of partition parallelism would either lead to incorrect results or the UDF query would fail. As a special case, should the UDF input fit in a single partition, the query would run but likely suboptimally. Currently, several UDFBench queries do not run 'as is' on Spark as they involve either blocking table UDFs (Q5, Q6) or non-parallel-safe UDFs (k-means Q10/Q11, tf/idf Q17, pivot Q19). Additionally, Spark does not support DML UDF queries (Q20, Q21). We tested Spark on local (for comparison with the other engines) and cluster (for distributed execution) modes. Spark offers an optional Arrow optimization to boost inter-operator data move via vectorization. This benefits a few queries (1.6x in Q2 with UDFs sub-type 3), but regresses many others (-4x in Q9 with UDFs sub-type 4). Next, we report results with Arrow disabled.

Data copy. MonetDB and DuckDB base their UDFs on NumPy and Pandas, and achieve zero-copy by passing arrays or dataframes to UDFs and returning results in the same format without data copy. PostgreSQL and SQLite invoke UDFs for each tuple, handle results as Python objects, which then transform into C representations.

UDF semantics. (a) *Multilingual queries* involving UDFs in multiple languages (Q3) are not supported in DuckDB and SQLite, while Spark supports UDFs in Python, Java, and Scala but not in SQL or C/C++. (b) *Iteration* is a common operation in modern UDFs; but iteration can also be implemented with *recursion.* And several works support recursive UDFs (e.g., [14]). UDFBench supports both implementations; e.g., queries Q10 and Q11 implement kmeans with iteration and recursion. However, in our experiments, we implement recursive UDFs using sub-functions inside the UDF body as the engines tested do not support natively recursive Python UDFs.

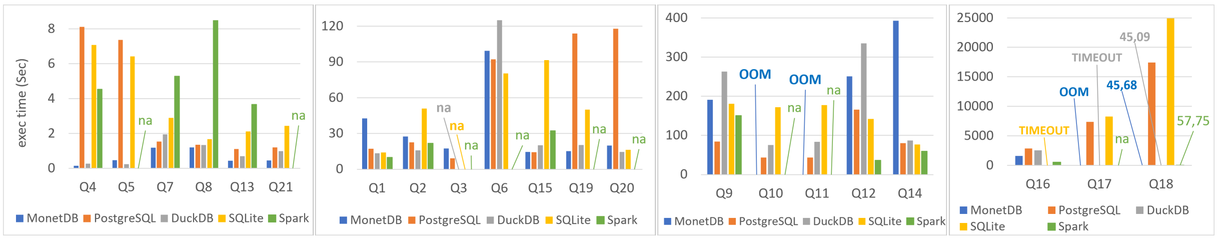


Figure 3: UDFBench query runtimes (in sec) for the large dataset grouped based on the magnitude of the measurements

Datasets. We ran the experiments on the small, medium, and large datasets described in Table 2. For space considerations, unless otherwise stated, we present results on the large dataset.

We investigate the key factors affecting UDF query execution and showcase how UDFBench can be used in other engines as well. We also present micro-experiments to detail issues related to UDF query performance, complexity, and expressiveness.

4.3 Performance analysis

4.3.1 UDF types. We run the UDFBench queries on the five engines (see Figure 3). Here, Spark runs on local mode for a fair comparison with the execution environment of the other engines. Queries of the first class QC1 (queries Q1-Q9) are suitable to study UDF performance. Scalar UDFs are examined in Q1-Q3 (multiple languages in Q3: C, Python, SQL). Since multi-thread parallelism in Python is limited by GIL, leveraging multi-processing parallelism (as PostgreSQL does) achieves greater speedups in these queries. Spark multi-process, tuple-at-a-time execution is faster than PostgreSQL (Q1, Q2) due to more aggressive partitioning. The aggregate UDFs in Q4 demonstrate the benefit of vectorized execution over tuple-at-a-time; here, MonetDB and DuckDB shine. Spark reads the entire dataset and directly applies the aggregation in one partition.

Most engines treat table UDFs of sub-type 4 (e.g., `combinations` in Q9) and sub-type 3 (e.g., `extractfromdate` in Q2) as blocking operations that limit parallelism. However, these UDFs could be parallelized as they process each tuple independently. For example, in Q9 PostgreSQL achieves nice parallelism. But Spark runs a suboptimal parallel scan on the input parquet file and uses a single partition for the aggregate. Table UDFs of sub-type 10 (e.g., `getstats` in Q5) are blocking and run sequentially in all engines. In this case, in-process, vectorized UDF execution is preferred due to seamless data exchange between the data engine and the UDF. Thus, MonetDB and DuckDB perform better than the other engines. On the other hand, executing this query type as out-process and tuple-at-a-time is sub-optimal, as it causes significant process overheads per tuple, which explains PostgreSQL weak performance here. Table UDFs of sub-types 8 and 9 (e.g., `file` and `output`, respectively in Q6) make the entire query to run sequentially in all engines. This query does not benefit from vectorized execution (e.g., it does not involve aggregation), so in-process, tuple-at-a-time execution achieves better results with SQLite being the fastest for this query. Spark is 'n/a' here due to the blocking table UDF `output`.

Query Q7 combines aggregate and scalar UDFs on top of a blocking, table UDF of sub-type 8 (`file`), which takes a single value (file path) and hence, makes the query run sequentially. Vectorization improves aggregation. Still, DuckDB faces an overhead caused by unnesting operations to support aggregates. MonetDB is faster than PostgreSQL, while SQLite is slower due to its aggregate implementation in Python. PostgreSQL good performance is due to its step function implementation for aggregates (e.g., 'accum' for `average`) [55]. Spark runs a suboptimal out-process single-partition execution.

Query Q8 is similar to Q7, but without the blocking table UDF, and highlights the pipeline parallelization of scalar and aggregate UDFs. As with Q1 and Q2, multi-processing allows PostgreSQL to achieve nice parallelization. Single-process, tuple-at-a-time SQLite is slower. Q8 involves a UDF (`jsoncount`) with a complex input type, which reduces the efficacy of vectorization as described in Section 2. Still, MonetDB and DuckDB achieve excellent execution due to vectorized processing. Spark parallelizes `jsoncount` but subperforms as it uses a single partition for the aggregate UDF.

4.3.2 Execution. Queries of the second class QC2 (10-13) integrate UDFs with relational operators, introducing challenges such as context switching, data conversion overheads, and optimization.

Q10 and Q11 implement kmeans with a blocking UDF of subtype 5 using Python scikit-learn and process Pandas dataframes either iterative (U35) or recursive (U36). These queries involve data exchange between a UDF and expensive subqueries (join, groupby). Vectorized execution with zero-copy exchange with the Pandas dataframes fits nicely this scenario, as DuckDB's performance indicates. Tuple-at-a-time implementation implicates dataframe creation with excessive data copies, and hence, SQLite is the slowest here. Parallel execution of the inner subquery benefits PostgreSQL runtime, despite the overhead of tranforming the subquery results to Pandas dataframes. Having a complex subquery as a parameter of a table UDF tricks MonetDB's optimizer to implement a cross product (crossjoin) and require excessive memory that eventually leads to out-of-memory (OOM) issues. If we split the query and the subquery, MonetDB employs algebra.join and performs as DuckDB.

Q12 merges expensive operators (group-by, aggregates) with scalar UDFs and a native aggregate (count). UDF parallelization with partitions helps PostgreSQL run faster than the other engines that fail to parallelize the query effectively due to GIL limitations. Spark shines due to aggressive parallelization; it uses 20 partitions for the scalar UDFs, while PostgreSQL with the same plan uses only 4.

Q13 involves 2 blocking table UDFs of sub-types 5 and 8 (`jsonparse` and `file`) with the outer table UDF (`jsonparse`) processing the results of an inner SQL subquery. Similarly to Q10 and Q11 zero-copy results into MonetDB and DuckDB being faster. Although Q13 does not involve translation into Pandas dataframes, PostgreSQL's out-process execution involves excessive data copies. Spark runs here with a single partition due to the table UDF sub-type 8 (`file`).

4.3.3 Optimization. Queries of the third class QC3 (14-19) involve complex interactions among UDFs and relational operators and introduce query optimization and execution challenges.

Q14 involves join operations with filters, group-by's, and UDFs (e.g., `jsonparse` UDF), and offers several optimization opportunities as described in Section 3.3. Indexing enhances this query, so SQLite shines with its automatic index creation, as PostgreSQL also does if indexes are explicitly created (our default PostgreSQL setup). Spark shines due to partitioning parallelism applied for `jsonparse`.

Q15 involves a not-in statement based on attributes produced by UDF pipelines. This stresses the optimizer; e.g., SQLite is slow due to a suboptimal nested-loop join selection. Similarly, Q18 executes a cross product on two tables produced by two UDF pipelines, and then runs an aggregation. Q18's `top` UDF (sub-type 7) is parallelizable per group, but current engines treat it as a black-box, limiting parallelism. Spark, as with Q7, is hurt by serial query execution due to the table UDF subtype 8 (`file`). Specifically, Q18 runs fully sequential and in Q15 only the not-in statement is parallelized. `file` is an example case where AQE could help. Spark cannot predict the output cardinality of `file` and runs the entire query on top of the UDF sequentially. Q15 and Q18 offer several currently unexplored opportunities for UDF optimization (e.g., UDF fusion [6]).

Q16 involves complex UDFs and SQL operations. The table UDF `combinations` (U31, sub-type 4) is parallelized in PostgreSQL (similar to Q9), however, a premature pull-up of `cleandate` into the case statement affects query performance. MonetDB and DuckDB, with less UDF invocations, run faster. Single-threaded SQLite is slower here. Spark shines due to a better plan choice (push-down `cleandate`) and process parallelism. We elaborate on missing optimization opportunities for this query in Section 4.6.

Q17 computes tf/idf scores and contains multiple aggregations and data grouping, which presumably make operator-at-a-time model preferable. This is true for small/medium datasets, where DuckDB and MonetDB outperform the tuple-at-a-time engines but with aggressive memory usage. However, in our environment, for the larger dataset, DuckDB and MonetDB fail with OOM errors.

Q19 highlights the advantage of operator-at-a-time engines to handle subqueries with zero-copy. In contrast, PostgreSQL struggles with tuple-at-a-time, inter-process communication, while SQLite benefits from in-process execution. Q19 involves the table UDF `pivot` (U38, sub-type 6), which is not parallelized by any engine.

4.3.4 DMLs. Queries of the fourth class QC4 (20-21) show the effect of UDFs in DML queries. We observe that the UDFs do not run in parallel for these queries, which affects their performance. Engines with indices enabled are also penalized for index maintenance. MonetDB is faster due to lazy materialization of updates [44].

4.3.5 Process size. The size of data processed by a UDF query is an indication of side-effects such as materialized outputs and data copies. We show aggregated results: (a) the average process size (MB) per engine for all queries (average), (b) the ratio p_a/p_m averaged for all queries, where p_a is an engine's process size for a query and p_m is the minimum process size among all engines for the same query (relative), and (c) the cardinality of p_m's per engine for all queries (count-min). UDFs in vectorized MonetDB and DuckDB process the least amount of data in 14 queries, and on average DuckDB processes half the data that MonetDB does. PostgreSQL and SQLite process overall over 20x the size of the data processed by the leanest engines, but still process the minimum size of data in 6 (Q3, Q6, Q9, Q13, Q15, Q21) and 1 (Q17) queries,

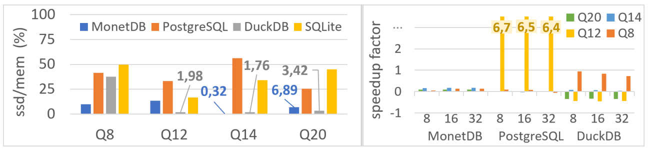


Figure 4: Disk/mem (left) and parallelism (right)

respectively. This is due to PostgreSQL's efficient use of indices. Q17 (tf/idf) failed to terminate for the large dataset on MonetDB and DuckDB as explained earlier. Spark presents similar behavior with PostgreSQL for the supported queries. However, it processes a larger data volume primarily due to broadcast joins that cause the same data to be processed multiple times across different workers. Also, memory-intensive compression codecs (e.g., Snappy) further increase memory consumption during processing.

process size	MonetDB	PostgreSQL	DuckDB	SQLite	Spark
average	2602.91	7305.68	1091,1	3886,43	5398,14
relative	4,77	21,45	6,06	23,31	80,31
count-min	5	6	9	1	0

As Spark shares similar architecture choices with the other engines (e.g., partition parallelism as PostgreSQL, albeit more aggressive) and it does not support many UDFBench queries, we do not include it in the rest of the analysis on single-node. We revisit it in Section 4.9 that explores UDF execution in a distributed setting.

4.4 Architecture choices

4.4.1 Disk vs. memory. Figure 4(left) shows the percentage (%) of improvement when queries from the 4 query classes (QC1-QC4) run in memory (`/dev/shm`) vs. on disk (ssd). PostgreSQL and SQLite achieve high speedups for in-memory runs (39% on average), due to the i/o overheads inherent to row-store, tuple-at-a-time execution. MonetDB and DuckDB achieve less diverse performance, with minimal speedups when running in memory (7.6% and 11%, respectively, on average), reflecting their efficient data access patterns.

4.4.2 Parallelism. SQLite is single-threaded so it is not applicable in this test. Figure 4(right) shows the execution speedup as we vary the number of threads: 8, 16, and 32. Multi-thread parallelism does not scale in Python with GIL. Query runtime is either slightly improved (MonetDB) or even degrades (DuckDB) due to excessive GIL contention. Multi-process parallelism achieves higher speedup with PostgreSQL: ~6.5x over single-threaded execution for 8 threads. Enabling additional threads does not affect the optimizer's choice. The update UDF query (Q20) does not improve with parallelism.

4.5 Stateful vs. stateless

We study the performance impact of stateful vs. stateless UDF variants with the `keywords` UDF (see Section 3.2) on the large dataset.

We tested `keywords` (U20) on three text columns of varying length with a simple UDF query '`select keywords(<col>) from <table>`' processing a varying size of non-null rows (`#rows`). The stateful UDF implementation compiles the pattern once at a global level and performs significantly faster (`speedup %`) than the stateless approach that compiles the pattern with each invocation. The gain increases with shorter text and high row counts. For columns with longer text, the UDF's inherent complexity dominates execution time, reducing the relative impact of the pattern compilation overhead.

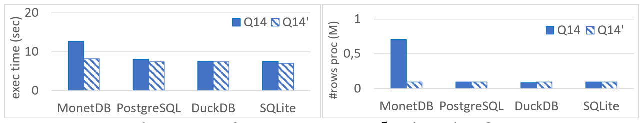


Figure 5: Operator reordering in Q14

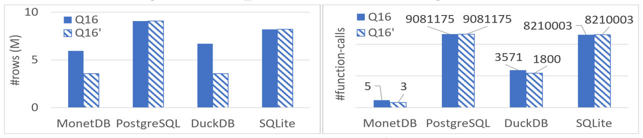


Figure 6: Operator reordering in Q16

The overhead of stateless execution, noted even in such a simple scenario, is critical in applications (e.g., text analytics) where the UDF involves heavy external libraries (e.g., nltk.download()).

	#rows	string len	speedup (%)
`keywords(abstract)`	1.3M	735	5.14
`keywords(title)`	3.8M	87	12.74
`keywords(fullname)`	9.9M	14	27.77

The results here are with SQLite, as PostgreSQL and MonetDB do not support stateful execution. For single UDF execution, stateless execution does not impact MonetDB or DuckDB as the pattern is compiled just once for the entire column. However, vectorized engines might benefit from stateful execution in queries with chains of UDFs, where a UDF could retain the state for an upstream UDF.

4.6 UDFs and query optimization

4.6.1 Operator reordering. Consider query Q14 and its variant Q14' described in Section 3.3. Figure 5 presents the query runtime and the number of rows processed by the `jsonparse` UDF on the small dataset. Both queries process the same number of rows 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. Notably, if `jsonparse` is implemented as a table UDF (sub-type 3) rather than a scalar UDF as used in this experiment, SQLite treats it as a black-box operator and does not perform reordering resulting in significant runtime difference (Q14 lasts 26s and Q14' only 12s, for the small dataset).

Let us now consider query Q16, which blends multiple UDFs with complex query logic, and investigate the impact of reordering the execution of the UDF operator `cleandate`. We test two semantically equivalent queries: (Q16): the original query (see Section 3.3), and (Q16'): in which the UDF is manually pushed down into the `form` clause to reduce the UDF invocations and, hopefully, to improve query performance. We measure the numbers of rows processed and the number of UDF calls in both queries. Figure 6 presents the results. SQLite and PostgreSQL exhibit identical plans for Q16 and Q16', indicating that both engines apply reordering. DuckDB and MonetDB process more rows in Q16 than Q16' and as their plans indicate they do not automatically reorder `cleandate`. In Q16', DuckDB processes the same number of rows as MonetDB but with more UDF calls, suggesting that it employs a batching mechanism.

4.6.2 UDF fusion. We investigate the impact of UDF fusion with Q16", a restructured version of Q16, in which we combine a series of UDFs and relational operators into fewer, fused UDFs to reduce the frequent context switches between the engine and the UDF execution environment. As described in Section 3.3, we create Q16" by manually fusing the upper segment of Q16 (portion A) into a single UDF. And we fuse the lower segment (portion C) into another UDF. Note that although the engines tested here do not support UDF fusion, related approaches have automated this process [6].

Figure 7 shows that Q16" outperforms Q16 in all four engines on the medium dataset. Q16 runtime varies across the engines due to UDF implementation differences. Although the relational operators of Q16 take about the same time (t-relop) in all engines but the single-threaded SQLite, the time spent in i/o (io-time), scalar (t-scalar) and table (t-table) UDFs varies. PostgreSQL runs sub-type 4, table UDFs (`combinations`) in parallel and pipelined, and it is 4x and 7x faster than DuckDB and MonetDB, respectively, where table UDFs materialize intermediate results. However, scalar UDFs are slower in PostgreSQL as the overhead of the excessive function calls (see Figure 6) is exacerbated by a suboptimal operator reordering that here, the other engines avoid. The runtime variance is smoother for Q16" that involves only two (fused) UDFs implemented similarly across all engines, which reduces the UDF integration overheads.

4.6.3 AQE. Adaptive query execution (AQE) deals with the lack of UDF statistics and may enable optimizations such as UDF reordering and fusion. Among the engines tested, only Spark supports AQE but to date mainly for physical join optimization. Q16 comprises a self-join over a table UDF. With AQE and careful tuning (e.g., autoBroadcastJoinThreshold = 1.5GB), after executing the table UDF `combinations`, Spark revises the running plan and achieves a 4.4x speedup over the static plan (550s -> 125s). AQE is a promising direction for UDF execution, but clearly more work is needed.

4.7 Compilation

The performance of UDF queries can be improved with JIT compilation, either at the UDF or the query levels [17]. UDFBench UDFs and queries offer opportunities for improving query performance with JIT compilation, should such option is offered by an engine. Let us consider Python UDFs as a case in point. Most engines are implemented in C and employ CPython to compile Python UDFs into intermediate bytecode. However, to date, most data engines do not JIT-compile the UDFs. Hence, in order to evaluate the impact of JIT compilation on UDF queries, we follow a solution proposed by YeSQL [19]: embed Python UDFs in C using CFFI, which supports both interpreted CPython and tracing JIT, such as PyPy. This setup allows us to define C UDFs with embedded Python, which can be executed on either CPython or PyPy across the four data engines.

In this experiment, we test performance for query Q1. Figure 8 compares UDF query execution with CPython and PyPy, and highlights the impact of JIT-compiled UDFs in query processing. In MonetDB, tracing JIT significantly boosts performance through parallelized UDF execution via multi-threading. However, in multi-threaded parallelism the GIL poses a constraint in both CPython and PyPy, as it involves a GIL lock/release per tuple in the C ↔ Python conversions. To address this, we apply a threading lock around the entire UDF bodies, which leads to substantial time reduction in CPython (`CPython-L`), where GIL-related contention during CFFI calls takes around 14 sec. The impact is lower in PyPy (`PyPy-L`), which releases the GIL more effectively. PostgreSQL is not affected by GIL limitations, as it uses multi-processes for parallelism, hence,

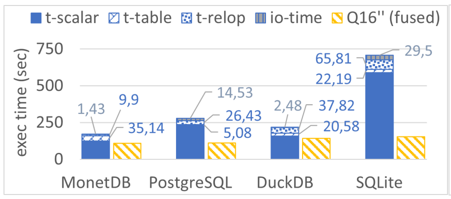


Figure 7: UDF fusion in Q16

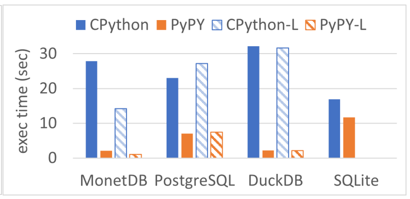


Figure 8: Compilation

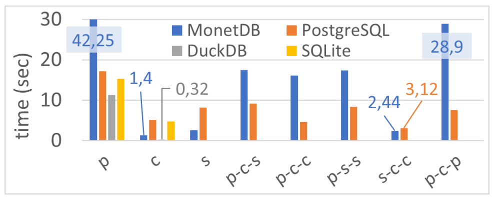


Figure 9: Languages: Python (p), C (c), SQL (s)

there is no further benefit from using a threading lock. Such a lock is also not useful for SQLite, which is single-threaded.

When executing UDFs with CPython in MonetDB and PostgreSQL, PostgreSQL achieves faster parallel execution thanks to its multi-processing that bypasses the GIL. However, with tracing JIT compilation and improved GIL handling MonetDB outperforms PostgreSQL. Even when running with locks (hence, resulting into serialized execution), MonetDB still surpasses PostgreSQL, as PostgreSQL's multi-processing, while improving parallelism, introduces additional overheads such as inter-process communication and copying UDF results back to the main process. DuckDB shows similar performance to MonetDB, however, it does not avoid GIL contentions with threading lock. Single-threaded, in-process SQLite also speedups its execution with JIT-compiled UDFs.

4.8 UDF programming language

In this experiment, we evaluate multilingual UDF queries. Query Q3 involves three scalar UDFs available in Python (p), C (c), and SQL (s). We test Q3 in eight variants: all UDFs in the same language (p or c or s), all UDFs in different languages (p-c-s), or UDFs in two languages (p-c-c, p-s-s, s-c-c, p-c-p). SQLite and DuckDB do not support multilingual queries or UDFs written in SQL; hence, only two variants apply on these engines: all UDFs either in Python or C. Figure 9 shows the impact of UDF language on query performance.

The query with C UDFs runs fast on all engines and faster on the vectorized MonetDB and DuckDB. It also runs in parallel in all engines but SQLite. Q3 with Python UDFs is overall slow and slower on MonetDB as its multi-threaded execution is limited by GIL, a problem that multi-process PostgreSQL avoids. MonetDB faces GIL-related slowdowns in all runs involving at least one Python UDF.

SQL UDFs run efficiently in MonetDB and PostgreSQL, but their runtime varies depending on their implementation. In Figure 9, the SQL UDFs in MonetDB are simple one-liners, but adding control flow statements (e.g., if/else) or variable declarations increases execution time significantly. For example, checking for null values with '`return ifthenelse(date,nullif(...),null);`' runs as a single atomic operation, while breaking it into separate statements results into multiple operators and intermediate materializations. In this micro-experiment, the former runs in 2.6s and the latter in 49.7s. Analyzing the query plans reveals that this 20x slowdown is due to excessive materializations in if/else statements.

4.9 UDF distributed execution

We tested UDF distributed execution on Spark in distributed mode (1 master, 3 workers), deployed on three VMs (28pt, 48GB DDR4). Here, Spark uses the same query plans as in local mode (Figure 3). Figure 10 shows the speedup when scaling from 1 to 2-3 workers.

As discussed in Section 4.2, 8 UDFBench queries do not run on

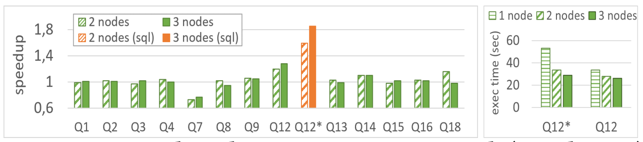


Figure 10: Distributed UDF execution on Spark (1 node = 1x)

Spark. Several of the supported queries (Q4, Q7, Q13, Q15, Q18) do not gain from partitioning parallelism due to the non-parallelizable UDF types they contain. Q7 is also hurt from slow `SparkFiles` file distribution. On the other hand, queries involving many relational operators (QC3) benefit from distributed execution with additional workers, while UDF-based queries (e.g., QC1) generally do not as the UDFs are treated as black boxes, preventing the application of inner UDF parallelism. Instead, only partitioned parallelism is used, where the UDF runs sequentially within each partition.

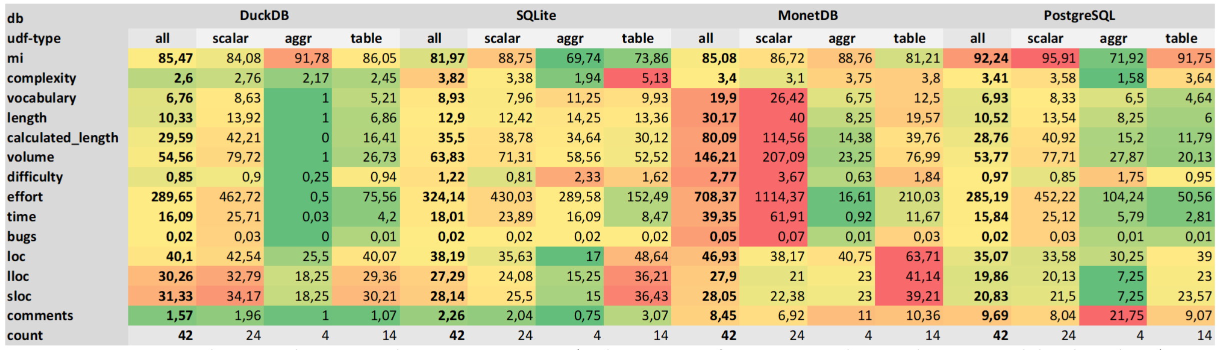
We study this further with Q12*, a variant of Q12 that uses only native Spark SQL without any UDFs. Q12* employs efficient parallelism and achieves ~2x speedup when scaling from 1 to 3 workers. Q12 exhibits faster execution on a single node (see Figure 10(right)) due to implicit operation fusion but limits parallelism in a distributed environment. In contrast, the SQL-based version decomposes computations into multiple operators, enabling better parallelization at the cost of additional interpretation overhead. The trade-off between encapsulating SQL operators within a single UDF and enabling parallel execution presents opportunities for optimizing UDF-heavy workloads, an interesting topic for future work.

Query Q3 comprises multilingual UDFs, which here are implemented in Scala (`extractday`), Java (`extractmonth`), and Python (`extractyear`), as these are the UDF languages supported in Spark. Multi-process execution in Spark works well for multilingual queries, as we did not observe any significant overhead due to different UDF languages such as those identified for other engines in Section 4.8.

4.10 Complexity and expressiveness

Besides performance, UDF developers also value highly easiness and maintainability. We perform static code analysis of the UDFBench UDFs as implemented in the various engines and evaluate their complexity and expressiveness using the software metrics discussed in Section 3.4.2. We do not include Spark here, as it does not support several UDFBench UDFs. The results are shown in Figure 11.

Although the UDFs follow the same logic, their implementation varies across the four engines. Overall, in terms of *maintenance* (mi - maintainability index), PostgreSQL is the most difficult and SQLite the simpler to maintain. In terms of *complexity* (i.e., cyclomatic complexity), SQLite requires the most complex code (especially, due to implementation of table UDFs), and DuckDB the simpler code. In terms of *size*, as it is expressed by the Halstead metrics (i.e., volume, length, etc.), MonetDB requires the lengthier and most convoluted code, as it involves conversions to NumPy for compute



db	DuckDB				SQLite				MonetDB				PostgreSQL			
udf-type	all	scalar	aggr	table	all	scalar	aggr	table	all	scalar	aggr	table	all	scalar	aggr	table
mi	**85,47**	84,08	91,78	86,05	**81,97**	88,75	69,74	73,86	**85,08**	86,72	88,76	81,21	**92,24**	95,91	71,92	91,75
complexity	**2,6**	2,76	2,17	2,45	**3,82**	3,38	1,94	5,13	**3,4**	3,1	3,75	3,8	**3,41**	3,58	1,58	3,64
vocabulary	**6,76**	8,63	1	5,21	**8,93**	7,96	11,25	9,93	**19,9**	26,42	6,75	12,5	**6,93**	8,33	6,5	4,64
length	**10,33**	13,92	1	6,86	**12,9**	12,42	14,25	13,36	**30,17**	40	8,25	19,57	**10,52**	13,54	8,25	6
calculated_length	**29,59**	42,21	0	16,41	**35,5**	38,78	34,64	30,12	**80,09**	114,56	14,38	39,76	**28,76**	40,92	15,2	11,79
volume	**54,56**	79,72	1	26,73	**63,83**	71,31	58,56	52,52	**146,21**	207,09	23,25	76,99	**53,77**	77,71	27,87	20,13
difficulty	**0,85**	0,9	0,25	0,94	**1,22**	0,81	2,33	1,62	**2,77**	3,67	0,63	1,84	**0,97**	0,85	1,75	0,95
effort	**289,65**	462,72	0,5	75,56	**324,14**	430,03	289,58	152,49	**708,37**	1114,37	16,61	210,03	**285,19**	452,22	104,24	50,56
time	**16,09**	25,71	0,03	4,2	**18,01**	23,89	16,09	8,47	**39,35**	61,91	0,92	11,67	**15,84**	25,12	5,79	2,81
bugs	**0,02**	0,03	0	0,01	**0,02**	0,02	0,02	0,02	**0,05**	0,07	0,01	0,03	**0,02**	0,03	0,01	0,01
loc	**40,1**	42,54	25,5	40,07	**38,19**	35,63	17	48,64	**46,93**	38,17	40,75	63,71	**35,07**	33,58	30,25	39
lloc	**30,26**	32,79	18,25	29,36	**27,29**	24,08	15,25	36,21	**27,9**	21	23	41,14	**19,86**	20,13	7,25	23
sloc	**31,33**	34,17	18,25	30,21	**28,14**	25,5	15	36,43	**28,05**	22,38	23	39,21	**20,83**	21,5	7,25	23,57
comments	**1,57**	1,96	1	1,07	**2,26**	2,04	0,75	3,07	**8,45**	6,92	11	10,36	**9,69**	8,04	21,75	9,07
count	**42**	24	4	14	**42**	24	4	14	**42**	24	4	14	**42**	24	4	14

Figure 11: Code complexity and expressiveness (color range: from green-low values to red-high values)

and intermediate results. Therefore, it also shows the highest effort and time to code. Note that although UDFs in MonetDB require the most lines of code (loc), the logical lines of code (lloc) is on par with the other engines. However, in PostgreSQL lloc is significantly lower indicating that it requires lesser executable "statements".

Figure 11 also reveals insights about the UDFs of UDFBench. The scalar UDFs implement string/text manipulation operations and are lengthier to code. However, MonetDB requires more complex code and DuckDB simpler code to implement the same logic. The aggregate UDFs have simpler implementations in DuckDB and PostgreSQL, require lengthier programs in SQLite, and are more complex in MonetDB. Interestingly, the complexity of aggregate UDFs in DuckDB is low but its maintainability index is high. This is due to our design that 'simulates' aggregate UDF in DuckDB implementing them as scalar UDFs in combination with SQL unnest (Section 4.2). The table UDFs are complex in all engines, but more in SQLite and MonetDB. Still, table UDFs in PostgreSQL are harder to maintain. Note that effectively in SQLite and DuckDB we measure our own design choices for implementing table UDFs (Section 4.2). Finally, MonetDB and PostgreSQL employ lengthier boilerplate code to create/register a UDF, which could help the UDF developer write less application-specific code. This is shown by the avg number of comment lines (~9 vs. ~2 comment lines in DuckDB/SQLite), as we commented the SQL and measure only the Python UDF code.

5 INSIGHTS AND OBSERVATIONS

Our analysis with UDFBench reveals insights and observations for improving UDF execution in data engines.

Parallelization. Multi-process UDF parallelism benefits queries with parallelizable UDFs. This is particularly useful for Python UDFs that suffer from GIL events. However, in distributed environments such as Spark, special care is required for partition parallelism.

Vectorization. Analytical queries excel in vectorized models processing aggregations on arrays or dataframes without data copies. Engines lacking native support for aggregate and table UDFs force developers to use potentially suboptimal workarounds (e.g., unnest). Some engines (e.g., Spark) employ vectorization via external frameworks (e.g., Arrow) to reduce communication overheads.

Optimization. Query optimization with UDFs remains challenging, as rule-based optimizers may misinterpret plans and unnecessarily employ costly operations like cross joins. Rewriting queries with temporary tables and indexes improves performance, especially in tuple-at-a-time models that struggle with joins on UDF-produced tables due to reliance on nested-loop joins. Additionally, the execution order of UDFs heavily impacts performance, with suboptimal placements in query plans causing UDFs to process unnecessary tuples and increase execution time.

Adaptive query execution (AQE). AQE presents opportunities for improved UDF query performance, although its potential is not yet fully realized. State-of-the-art systems utilize AQE mainly for physical optimization of joins, leaving other aspects of UDF query execution (e.g., order in UDF chains) largely unexplored.

UDF types. Currently, most SQL engines have limited support for table UDFs. Several UDF sub-types are not currently considered, which leads to sub-optimal or even incompatible UDF queries.

Security. Unrestricted UDF operations (e.g., `fwrite`, `fopen`) highlight the need for enhanced security. Out-process UDF execution mitigates the risk to some extent and enhances fault isolation at the cost of performance overheads for non-parallelizable UDFs, which in general run faster with in-process execution.

Unexplored aspects. Future systems should explore aspects such as sharing computations across UDFs, serverless structures (e.g., dictionaries) for stateful execution with lambda functions, fusion of UDFs and relational operators, JIT compilation, and enhanced query optimization strategies. Additionally, mechanisms for dynamic UDF push-down or push-up could reduce context-switching overhead and better integrate UDFs with the underlying query engine. Addressing such challenges would be essential for achieving performance gains and robust security in UDF query execution.

6 CONCLUSIONS

Extending declarative SQL with functional capabilities is essential to support modern applications and to some extent, could be a path to a new market for database products. To advance toward a new generation of data engines, where UDFs (such as AI operators) are treated as first-class citizens, we must first systematically identify the strengths and limitations of current solutions. To facilitate this, we introduced the UDFBench, a tool designed to help the community standardize the evaluation and comparison of database engines that support multi-lingual UDFs in SQL queries, and to provide a testbed for a systematic analysis of their capabilities.

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. APSW documentation, available at: https://rogerbinns.github.io/apsw.

[2] Samuel Arch, Yuchen Liu, Todd C. Mowry, Jignesh M. Patel, and Andrew Pavlo. 2024. The Key to Effective UDF Optimization: Before Inlining, First Perform Outlining. *Proc. VLDB Endow.* 18, 1 (2024), 1–13.

[3] Matthias Boehm, Berthold Reinwald, Dylan Hutchison, Prithviraj Sen, Alexandre V. Evfimievski, and Niketan Pansare. 2018. On Optimizing Operator Fusion Plans for Large-Scale Machine Learning in SystemML. *Proc. VLDB Endow.* 11, 12 (2018), 1755–1768.

[4] Nicolas Bruno and Surajit Chaudhuri. 2005. Flexible Database Generators. In *VLDB*.

[5] Michael J. Cafarella and Christopher Ré. 2010. Manimal: Relational Optimization for Data-Intensive Programs. In *WebDB*.

[6] Konstantinos Chasialis, Theoni Palaiologou, Yannis Foufoulas, Alkis Simitsis, and Yannis E. Ioannidis. 2024. QFusor: A UDF Optimizer Plugin for SQL Databases. In *40th IEEE International Conference on Data Engineering, ICDE 2024, Utrecht, The Netherlands, May 13-16, 2024*. IEEE, 5457–5460.

[7] Surajit Chaudhuri and Kyuseok Shim. 1993. Query Optimization in the Presence of Foreign Functions. In *VLDB*. 529–542.

[8] Surajit Chaudhuri and Kyuseok Shim. 1996. Optimization of Queries with User-defined Predicates. In *VLDB'96*. 87–98.

[9] Surajit Chaudhuri and Kyuseok Shim. 1999. Optimization of Queries with User-Defined Predicates. *ACM Trans. Database Syst.* 24, 2 (1999), 177–228.

[10] Andrew Crotty, Alex Galakatos, Kayhan Dursun, Tim Kraska, Ugur Çetintemel, and Stanley B. Zdonik. 2015. Tupleware: "Big" Data, Big Analytics, Small Clusters. In *CIDR*. www.cidrdb.org.

[11] Harshad Deshmukh, Hakan Memisoglu, and Jignesh M. Patel. 2017. Adaptive Concurrent Query Execution Framework for an Analytical In-Memory Database System. In *2017 IEEE International Congress on Big Data, BigData Congress 2017, Honolulu, HI, USA, June 25-30, 2017*, George Karypis and Jia Zhang (Eds.). IEEE Computer Society, 23–30.

[12] Djellel Eddine Difallah, Andrew Pavlo, Carlo Curino, and Philippe Cudré-Mauroux. 2013. OLTP-Bench: An Extensible Testbed for Benchmarking Relational Databases. *PVLDB* 7, 4 (2013), 277–288.

[13] Markus Dreseler, Martin Boissier, Tilmann Rabl, and Matthias Uflacker. 2020. Quantifying TPC-H Choke Points and Their Optimizations. *Proc. VLDB Endow.* 13, 8 (2020), 1206–1220.

[14] Christian Duta and Torsten Grust. 2020. Functional-Style SQL UDFs With a Capital 'F'. In *SIGMOD*. 1273–1287.

[15] Christian Duta, Denis Hirn, and Torsten Grust. 2020. Compiling PL/SQL Away. In *CIDR*.

[16] Tim Fischer, Denis Hirn, and Torsten Grust. 2024. SQL Engines Excel at the Execution of Imperative Programs. *Proc. VLDB Endow.* 17, 13 (2024), 4696–4708.

[17] Yannis Foufoulas and Alkis Simitsis. 2023. Efficient Execution of User-Defined Functions in SQL Queries. *Proc. VLDB Endow.* 16, 12 (2023), 3874–3877.

[18] Yannis Foufoulas and Alkis Simitsis. 2023. User-Defined Functions in Modern Data Engines. In *39th IEEE International Conference on Data Engineering, ICDE 2023, Anaheim, CA, USA, April 3-7, 2023*. IEEE, 3593–3598.

[19] 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.

[20] Kai Franz, Samuel Arch, Denis Hirn, Torsten Grust, Todd C. Mowry, and Andrew Pavlo. 2024. Dear User-Defined Functions, Inlining isn't working out so great for us. Let's try batching to make our relationship work. Sincerely, SQL. In *14th Conference on Innovative Data Systems Research, CIDR 2024, Chaminade, HI, USA, January 14-17, 2024*.

[21] Henning Funke, Jan Mühlig, and Jens Teubner. 2022. Low-latency query compilation. *VLDB J.* 31, 6 (2022), 1171–1184.

[22] Philipp Marian Grulich, Steffen Zeuch, and Volker Markl. 2021. Babelfish: Efficient execution of polyglot queries. *Proceedings of the VLDB Endowment* 15, 2 (2021), 196–210.

[23] Tim Gubner and Peter A. Boncz. 2022. Excalibur: A Virtual Machine for Adaptive Fine-grained JIT-Compiled Query Execution based on VOILA. *Proc. VLDB Endow.* 16, 4 (2022), 829–841.

[24] Surabhi Gupta and Karthik Ramachandra. 2021. Procedural Extensions of SQL: Understanding their usage in the wild. *Proc. VLDB Endow.* 14, 8 (2021), 1378–1391.

[25] Stefan Hagedorn, Steffen Kläbe, and Kai-Uwe Sattler. 2021. Putting Pandas in a Box. In *CIDR*.

[26] Maurice H. Halstead. 1977. *Elements of Software Science*. Elsevier, New York.

[27] Joseph M. Hellerstein and Jeffrey F. Naughton. 1996. Query Execution Techniques for Caching Expensive Methods. In *SIGMOD*. 423–434.

[28] Joseph M. Hellerstein and Michael Stonebraker. 1993. Predicate Migration: Optimizing Queries with Expensive Predicates. In *SIGMOD*. 267–276.

[29] Fabian Hueske, Mathias Peters, Aljoscha Krettek, Matthias Ringwald, Kostas Tzoumas, Volker Markl, and Johann-Christoph Freytag. 2013. Peeking into the optimization of data flow programs with MapReduce-style UDFs. In *ICDE*. 1292–1295.

[30] Petar Jovanovic, Alkis Simitsis, and Kevin Wilkinson. 2014. BabbleFlow: a translator for analytic data flow programs. In *SIGMOD*. 713–716.

[31] Petar Jovanovic, Alkis Simitsis, and Kevin Wilkinson. 2014. Engine independence for logical analytic flows. In *ICDE*. 1060–1071.

[32] Michael Jungmair and Jana Giceva. 2023. Declarative Sub-Operators for Universal Data Processing. *Proc. VLDB Endow.* 16, 11 (2023), 3461–3474.

[33] Gaurav Tarlok Kakkar, Jiashen Cao, Aubhro Sengupta, Joy Arulraj, and Hyesoon Kim. 2024. Hydro: Adaptive Query Processing of ML Queries. *CoRR* abs/2403.14902 (2024).

[34] Ken Kennedy and Kathryn S. McKinley. 1993. Maximizing Loop Parallelism and Improving Data Locality via Loop Fusion and Distribution. In *LCPC*, Vol. 768. 301–320.

[35] 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. *PVLDB* 11, 13 (2018), 2209–2222.

[36] Steffen Kläbe, Bobby DeSantis, Stefan Hagedorn, and Kai-Uwe Sattler. 2022. Accelerating Python UDFs in Vectorized Query Execution. In *Proceedings of the Annual Conference on Innovative Data Systems Research*. CIDR.

[37] Siu Kwan Lam, Antoine Pitrou, and Stanley Seibert. 2015. Numba: A LLVM-Based Python JIT Compiler. In *LLVM-SC*.

[38] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter A. Boncz, Alfons Kemper, and Thomas Neumann. 2015. How Good Are Query Optimizers, Really? *Proc. VLDB Endow.* 9, 3 (2015), 204–215.

[39] Zhan Li, Olga Papaemmanouil, and Mitch Cherniack. 2016. OptMark: A Toolkit for Benchmarking Query Optimizers. In *Proceedings of the 25th ACM International Conference on Information and Knowledge Management, CIKM 2016, Indianapolis, IN, USA, October 24-28, 2016*. ACM, 2155–2160.

[40] Rui Liu, Jun Hyuk Chang, Riki Otaki, Zhe Heng Eng, Aaron J. Elmore, Michael J. Franklin, and Sanjay Krishnan. 2024. Towards Resource-adaptive Query Execution in Cloud Native Databases. In *14th Conference on Innovative Data Systems Research, CIDR 2024, Chaminade, HI, USA, January 14-17, 2024*. www.cidrdb.org.

[41] Paolo Manghi, Claudio Atzori, Alessia Bardi, Miriam Baglioni, Harry Dimitropoulos, Sandro La Bruzzo, Ioannis Foufoulas, Andrea Mannocci, Marek Horst, Katerina Iatropoulou, Argiro Kokogiannaki, Michele De Bonis, Michele Artini, Antonis Lempesis, Alexandros Ioannidis, Natalia Manola, Pedro Principe, Thanasis Vergoulis, Serafeim Chatzopoulos, and Dimitris Pierrakos. 2024. *OpenAIRE Graph Dataset*. https://doi.org/10.5281/zenodo.10488385

[42] T.J. McCabe. 1976. A Complexity Measure. *IEEE Transactions on Software Engineering* SE-2, 4 (1976), 308–320.

[43] Microsoft. 2024. Transact-SQL, available at: https://learn.microsoft.com/en-us/sql/t-sql.

[44] MonetDB. 2024. Transaction Schema, available at: https://www.monetdb.org/documentation-Aug2024/admin-guide/transaction-schema.

[45] Thomas Neumann. 2021. Evolution of a Compiling Query Engine. *Proc. VLDB Endow.* 14, 12 (2021), 3207–3210.

[46] T. Neumann, S. Helmer, and G. Moerkotte. 2005. On the optimal ordering of maps and selections under factorization. In *ICDE*. 490–501.

[47] Thomas Neumann, Sven Helmer, and Guido Moerkotte. 2006. On the Optimal Ordering of Maps, Selections, and Joins Under Factorization. In *BNCOD*. 115–126.

[48] Paul W. Oman and Jack R. Hagemeister. 1992. Metrics for assessing a software system's maintainability. In *Proceedings of the Conference on Software Maintenance, ICSM 1992, Orlando, FL, USA, 9-12 November, 1992*. IEEE Computer Society, 337–344.

[49] Pat O'Neil, Betty O'Neil, and Xuedong Chen. 2009. The Star Schema Benchmark (SSB). (2009).

[50] Patrick E. O'Neil, Elizabeth J. O'Neil, Xuedong Chen, and Stephen Revilak. 2009. The Star Schema Benchmark and Augmented Fact Table Indexing. In *Performance Evaluation and Benchmarking, First TPC Technology Conference, TPCTC 2009, Lyon, France, August 24-28, 2009, Revised Selected Papers (Lecture Notes in Computer Science, Vol. 5895)*. Springer, 237–252.

[51] OpenAire. 2025. Text Exploration and Analysis. Available at: https://github.com/openaire/iis/tree/master/iis-wf/iis-wf-referenceextraction/src/main/resources/eu/dnetlib/iis/wf/referenceextraction.

[52] Oracle. 2024. Getting Started With PL/SQL, available at: https://www.oracle.com/database/technologies/appdev/plsql.html.

[53] Shoumik Palkar, James Thomas, Deepak Narayanan, Pratiksha Thaker, Rahul Palamuttam, Parimajan Negi, Anil Shanbhag, Malte Schwarzkopf, Holger Pirk, Saman Amarasinghe, Samuel Madden, and Matei Zaharia. 2018. Evaluating end-to-end optimization for data analytics applications in weld. In *Proceedings of the VLDB Endowment*. VLDB, 1002–1015.

[54] PostgreSQL. 2022. PL/pgSQL, SQL Procedural Language. Available at: https://www.postgresql.org/docs/current/plpgsql.html.

[55] PostgreSQL. 2024. PostgreSQL documentation: User-Defined Aggregates, available at: https://www.postgresql.org/docs/current/xaggr.html.

[56] Fotis Psallidas, Yiwen Zhu, Bojan Karlas, Matteo Interlandi, Avrilia Floratou, Konstantinos Karanasos, Wentao Wu, Ce Zhang, Subru Krishnan, Carlo Curino, and Markus Weimer. 2019. Data Science through the looking glass and what we

found there. *CoRR* abs/1912.09536 (2019). http://arxiv.org/abs/1912.09536

[57] PyPy. 2024. Radon, available at: https://pypi.org/project/radon.

[58] Python. 2022. Global interpreter lock. Available at: https://wiki.python.org/moin/GlobalInterpreterLock.

[59] Mark Raasveld and Hannes Mühleisen. 2016. Vectorized UDFs in Column-Stores. In *Proceedings of the 28th International Conference on Scientific and Statistical Database Management*. SSDBM, 1–12.

[60] Karthik Ramachandra, Kwanghyun Park, K. Venkatesh Emani, Alan Halverson, César A. Galindo-Legaria, and Conor Cunningham. 2017. Froid: Optimization of Imperative Programs in a Relational Database. *PVLDB* 11, 4 (2017), 432–444.

[61] Viktor Rosenfeld, René Müller, Pinar Tözün, and Fatma Özcan. 2017. Processing Java UDFs in a C++ environment. In *SoCC*. 419–431.

[62] Maximilian E. Schüle, Jakob Huber, Alfons Kemper, and Thomas Neumann. 2020. Freedom for the SQL-Lambda: Just-in-Time-Compiling User-Injected Functions in PostgreSQL. In *SSDBM*. 6:1–6:12.

[63] SQL Server. 2024. Intelligent query processing, available at: https://learn.microsoft.com/en-us/sql/relational-databases/performance/intelligent-query-processing-details.

[64] Hesam Shahrokhi, Callum Groeger, Yizhuo Yang, and Amir Shaikhha. 2023. Efficient Query Processing in Python Using Compilation. In *SIGMOD*. 199–202.

[65] Hesam Shahrokhi and Amir Shaikhha. 2023. Building a Compiled Query Engine in Python. In *SIGPLAN*. ACM, 180–190.

[66] Moritz Sichert and Thomas Neumann. 2022. User-defined operators: efficiently integrating custom algorithms into modern databases. *Proceedings of the VLDB Endowment* 15, 5 (2022), 1119–1131.

[67] Moritz Sichert and Thomas Neumann. 2022. User-Defined Operators: Efficiently Integrating Custom Algorithms into Modern Databases. *PVLDB* 15, 5 (2022), 1119–1131.

[68] Varun Simhadri, Karthik Ramachandra, Arun Chaitanya, Ravindra Guravannavar, and S. Sudarshan. 2014. Decorrelation of user defined function invocations in queries. In *ICDE*. 532–543.

[69] Alkis Simitsis, Spiros Skiadopoulos, and Panos Vassiliadis. 2023. The History, Present, and Future of ETL Technology (invited). In *Proceedings of the 25th International Workshop on Design, Optimization, Languages and Analytical Processing of Big Data (DOLAP)*, Vol. 3369. 3–12.

[70] Alkis Simitsis, Panos Vassiliadis, Umeshwar Dayal, Anastasios Karagiannis, and Vasiliki Tziovara. 2009. Benchmarking ETL Workflows. In *Performance Evaluation and Benchmarking, First TPC Technology Conference, TPCTC 2009, Lyon, France, August 24-28, 2009, Revised Selected Papers (Lecture Notes in Computer Science, Vol. 5895)*. Springer, 199–220.

[71] Alkis Simitsis, Panos Vassiliadis, Manolis Terrovitis, and Spiros Skiadopoulos. 2005. Graph-Based Modeling of ETL Activities with Multi-level Transformations and Updates. In *Data Warehousing and Knowledge Discovery (Lecture Notes in Computer Science, Vol. 3589)*. Springer, 43–52.

[72] Alkis Simitsis, Kevin Wilkinson, Umeshwar Dayal, and Meichun Hsu. 2013. HFMS: Managing the lifecycle and complexity of hybrid analytic data flows. In *29th IEEE International Conference on Data Engineering, ICDE 2013, Brisbane, Australia, April 8-12, 2013*. IEEE Computer Society, 1174–1185.

[73] Leonhard F Spiegelberg, Rahul Yesantharao, Malt Schwarzkopf, and Tim Kraska. 2021. Tuplex: Data Science in Python at Native Code Speed. In *Proceedings of the 2021 International Conference on Management of Data*. SIGMOD, 1718–1731.

[74] SQLITE. 2022. Application-Defined SQL Functions Available at: https://www.sqlite.org/appfunc.html.

[75] MS Visual Studio. 2024. Code metrics - Maintainability index range and meaning, available at: https://learn.microsoft.com/en-us/visualstudio/code-quality/code-metrics-maintainability-index-range-and-meaning?view=vs-2022.

[76] Transaction Processing Performance Council. 2024. Active TPC Benchmarks, available at: https://www.tpc.org/information/benchmarks5.asp.

[77] UDFBench. 2025. Code repository, available at: https://github.com/athenarc/UDFBench.

[78] Panos Vassiliadis, Anastasios Karagiannis, Vasiliki Tziovara, and Alkis Simitsis. 2007. Towards a Benchmark for ETL Workflows. In *Proceedings of the Fifth International Workshop on Quality in Databases, QDB 2007, at the VLDB 2007 conference, Vienna, Austria, September 23, 2007*. 49–60.

[79] Maryann Xue, Yingyi Bu, Abhishek Somani, Wenchen Fan, Ziqi Liu, Steven Chen, Herman Van Hovell, Bart Samwel, Mostafa Mokhtar, Rk Korlapati, Andy Lam, Yunxiao Ma, Vuk Ercegovac, Jiexing Li, Alexander Behm, Yuanjian Li, Xiao Li, Sriram Krishnamurthy, Amit Shukla, Michalis Petropoulos, Sameer Paranjpye, Reynold Xin, and Matei Zaharia. 2024. Adaptive and Robust Query Execution for Lakehouses At Scale. *Proc. VLDB Endow.* 17, 12 (2024), 3947–3959.

[80] Kenichi Yajima, Hiroyuki Kitagawa, Kazunori Yamaguchi, Nobuo Ohbo, and Yuzuru Fujiwara. 1991. Optimization of Queries Including ADT Functions. In *DASFAA*, Vol. 2. 366–373.

[81] Cong Yan, Yin Lin, and Yeye He. 2023. Predicate Pushdown for Data Science Pipelines. *Proc. ACM Manag. Data* 1, 2 (2023), 136:1–136:28.