YeSQL: "You extend SQL" with Rich and Highly Performant User-Defined Functions in Relational Databases

Yannis Foufoulas
U. of Athens, Athena R.C.
johnfouf@di.uoa.gr

Alkis Simitsis
Athena Research Center
alkis@athenarc.gr

Lefteris Stamatogiannakis
University of Athens
estama@gmail.com

Yannis Ioannidis
U. of Athens, Athena R.C.
yannis@di.uoa.gr

ABSTRACT

The diversity and complexity of modern data management applications have led to the extension of the relational paradigm with syntactic and semantic support for User-Defined Functions (UDFs). Although well-established in traditional DBMS settings, UDFs have become central in many application contexts as well, such as data science, data analytics, and edge computing. Still, a critical limitation of UDFs is the impedance mismatch between their evaluation and relational processing. In this paper, we present YeSQL, an SQL extension with rich UDF support along with a pluggable architecture to easily integrate it with either server-based or embedded database engines. YeSQL currently supports Python UDFs fully integrated with relational queries as scalar, aggregator, or table functions. Key novel characteristics of YeSQL include easy implementation of complex algorithms and several performance enhancements, including tracing JIT compilation of Python UDFs, parallelism and fusion of UDFs, stateful UDFs, and seamless integration with a database engine. Our experimental analysis showcases the usability and expressiveness of YeSQL and demonstrates that our techniques of minimizing context switching between the relational engine and the Python VM are very effective and achieve significant speedups up to 68x in common, practical use cases compared to earlier approaches and alternative implementation choices.

PVLDB Reference Format:
Yannis Foufoulas, Alkis Simitsis, Lefteris Stamatogiannakis, and Yannis Ioannidis. YeSQL: "You extend SQL" with Rich and Highly Performant User-Defined Functions in Relational Databases. PVLDB, 15(10): 2270 - 2283, 2022.
doi:10.14778/3547305.3547328

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

1 INTRODUCTION

Modern trends in data processing are characterized by a diversity of data sources and complex processing tasks running on large volumes of data. This falls naturally within the scope of relational databases, which are extremely powerful data storage and processing engines. Many such tasks, however, cannot be expressed in SQL and require additional expressive power, achieved via User-Defined Functions (*UDFs*), typically written in C++, Java, or Python, which is the focus of this work. Python is the UDF language of choice for many data analysts and data scientists, as it is easy to learn and convenient to use, offering a rich feature set for data-intensive tasks, including automatic memory management, dynamic typing, and an extensive module ecosystem.

Proceedings of the VLDB Endowment, Vol. 15, No. 10 ISSN 2150-8097.
doi:10.14778/3547305.3547328

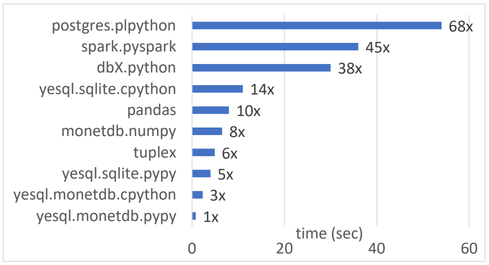


Figure 1: Performance of a scalar UDF on a string column

Python UDFs are supported by most data processing systems, but currently have several limitations on their *usability*, *expressiveness*, and *performance* [29, 37, 68]. For example, MonetDB [59] natively supports vectorized Python UDFs using Numpy[49], but these require static definition of their return schema, as only table UDFs may return multiple rows. In terms of performance, no standard booster such as a Just-In-Time (JIT) compiler is used. The functions run on Python's interpreter and in case Numpy does not support the desired functionality, extra data structure transformations must be applied in order to proceed in CPython. Likewise, PostgreSQL [53] also supports Python UDFs without a JIT compiler. UDFs can be stateful using a dictionary that is passed as a parameter, but looking up the dictionary adds overhead. The functions are not fully polymorphic; it is possible to create a function that specifies its output based on the types of its input but not based on the data, e.g., a table function that parses and imports an external file.

In this paper, we present *YeSQL*[1], an SQL extension and its implementation that provides more *usable*, more *expressive*, and more *perfomant* Python UDFs and can be integrated into both server-based and embedded DBMSs. It enriches SQL with a functional syntax that unifies the expression of relational and user-defined functionality and optimizes the execution of both in a seamless fashion, assigning processing tasks to the DBMS or the UDF host language VM accordingly and employing efficient low-level implementation techniques.

In more detail, the contributions of this work come in the form of the following YeSQL characteristics:

Usability and expressiveness. YeSQL extends SQL with an alternative, equivalent syntax that affords compact expressions of many

[1]Stands for "You extend SQL" and is pronounced "YES Q L".

relational queries and also facilitates the uniform expression of complex compositions of multiple UDFs and relational functions. This reduces significantly the programmer's time needed to compose a new algorithm or pipeline. Key characteristics of the YeSQL language that enhance usability and expressiveness include (a) stateful, parametric, and polymorphic UDFs, (b) dynamically typed UDFs, (c) scalar and aggregate UDFs returning arbitrary table forms, and (d) UDF pipelining.

Performance. YeSQL improves Python UDF performance by reducing the main UDF-call bottlenecks: (i) data conversions and copies when UDF input and output is translated from and to SQL and (ii) overheads of running complex analysis on CPython's interpreter (i.e., the default and most widely used implementation of the Python language). It does this by employing (a) seamless data exchange between the UDF and the DBMS, (b) JIT-compiled UDFs, (c) UDF parallelization, (d) stateful UDFs, and (e) UDF fusion. The latter combines multiple UDFs into one, thereby reducing data conversions, copies, and context switches between different execution environments. Moreover, it allows different UDFs to run in the same execution trace, reaping the benefits of tracing JIT[2]. YeSQL uses PyPy [6] as its tracing JIT compiler and CFFI [61] to interact with the C language. The latter's support of both CPython and PyPy enables execution of UDFs in C with either one transparently. UDF fusion also opens up query optimization opportunities, which are part of our future work.

For a sneak preview of the performance benefits of YeSQL, we present an example comparison of a UDF running on PostgreSQL with PL/Python, MonetDB/NumPy, SQLite, a popular commercial distributed column-store DBMS (denoted dbX), Tuplex [68], Spark/PySpark, Pandas, and YeSQL extensions to MonetDB and SQLite with CPython and PyPy. Figure 1 shows that JIT-compiled YeSQL implementations allow MonetDB and SQLite to run 6x to 68x faster than the other candidates (bars show time, labels show gain). We present more details and extensive experiments in Section 6.

Portability. YeSQL is designed to work in synergy with existing systems. It is implemented on top of SQLITE API, originally introduced in SQLite, but now commonly used in embedded DBMSs such as BerkeleyDB, DuckDB, and Monetdbe. The YeSQL architecture extends this functionality to server-based architectures and is also compatible with works such as Hustle [55], a recent effort that introduces a query acceleration path for analytics in SQLite3, utilizing Apache Arrow[1] as an in-memory columnar storage layer to transform SQLite into a hybrid database system.

Deployment. YeSQL is currently used in production by OpenAIRE[3], a technical infrastructure co-designed and co-developed in the context of a consortium of 65 European universities, research centers, and other institutions, offering services that were invoked 42M times last year in the context of 1M visits. OpenAIRE data scientists use YeSQL daily to harvest research output from >1000 connected data providers and classify [22], text mine Open Access publications, and extract links to funders [21], software, citations [19], datasets, bioentities, and other information. To date, 129M publications, 2M datasets, 85K research software artifacts, and 1.5M research projects from 23 different national and international funders have been harvested using over 150 YeSQL UDFs.

[2]This is more evident with trace-compilers that produce optimized machine code at the level of traces (a series of instructions commonly executed in sequence). Exposing longer sequences of instructions enables more optimization in the Python execution.
[3]https://www.openaire.eu, https://explore.openaire.eu

Based on the above, we believe that YeSQL is a significant step forward in the direction of enhancing the usability and improving the performance of user-defined functionality inside DBMSs. Its design and implementation can serve as good starting points for future data processing environments.

The structure of the paper is as follows. Section 2 reviews prior art and its advantages and limitations. Section 3 presents an overview of YeSQL. Section 4 describes the YeSQL language, its functionality and syntax support. Section 5 explains our design choices to boost YeSQL performance. Section 6 demonstrates the large potential of YeSQL through a series of experiments, micro-benchmarks and a comparison with the state of the art. Section 7 concludes the paper and discusses potential future directions.

2 RELATED WORK

Recent data processing trends have favored moving much of the computation inside user-defined operators. Many data processing systems support Python UDFs. Moreover, several works target to speed up Python via compilation or translation. Here, we discuss prior work on optimizing Python code such as compilers and transpilers, and data processing systems with support of Python UDFs.

2.1 Python Compilers and Transpilers

Compilers. GraalPython [26] is an experimental implementation of Python3. It uses the Truffle [33] language interpreter to generate JVM bytecode for JIT compilation. It supports Scipy [64] and optimizes pure Python code (i.e., without C extensions) when embedded into Java. Numba [39] JIT-compiles Python only when it can infer the data types, and targets array-structured data from libraries such as NumPy. Thus, it does not properly work with dynamic typed Python neither it supports importing external libraries. Pyston [43] implements a tracing JIT Python compiler using LLVM, but it is only 20% faster than CPython in real scenarios [44].

Transpilers. These works translate Python into other languages. Cython [2] translates Python code into C. However, it requires declaring the data types to obtain significant optimization benefits, thus sacrificing much of its performance benefits if used with existing Python code. Nuitka [48] compiles Python to C++, however, it suffers from slow compilation time, which is important in case of using it to create and execute UDFs in a data processing system.

Our experimental evaluation demonstrates that YeSQL offers superior performance than popular compiler and transpiler approaches for boosting Python UDFs (we discuss this in Section 6.3.1).

2.2 Data Processing Systems

Most commercial and open source data processing systems, including the open-source MonetDB and PostgreSQL, support Python UDFs. However, as we discussed, these implementations currently have significant restrictions w.r.t expressiveness and performance. The research community has recognized the problem and has proposed some approaches to improve Python UDF performance.

Tuplex [68] is a research prototype of a JIT compiler focusing on Python, with excellent performance that outperforms several popular data processing system. Tuplex lacks expressiveness features that would render the approach more practical in many real-world applications. For example, functions are not fully parametric, they get a dictionary parameter containing the whole tuple and the desired column is selected inside the UDF. They are stateless and without side-effects, and do not support exception handling. The latter would also be a performance issue, as exceptions are handled in a different execution context with Python's interpreter, often making it a significant overhead when processing dirty data. Still, we find Tuplex [68] as the work most relevant to ours. Our focus is on improving Python UDFs in DBMS, but even so we find it challenging to compare against an optimized, end-to-end tracing JIT compiler that comes without any (positive or negative) DBMS luggage. Besides its shortcomings related to expressiveness and usability, we also have distinct differences in the architecture design and implementation. We discuss these in the rest of the paper and present a thorough comparison in the experiments, using the dataset and queries used by Tuplex [68].

Kläbe et al. [37] compare different compilation frameworks for Python UDFs and analyze techniques to optimize their execution inside a DBMS. This work focuses on assessing the following techniques (a) compilation using different frameworks (i.e., Cython, Nuitka, Numba), (b) vectorized UDF execution [36], (c) parallelization via multiprocessing, and compares these compilation frameworks against interpreted CPython UDFs. The initial prototype integrates the Cython compilation framework into the Actian Vector database. Although the work is still at a very early stage, a preliminary evaluation of end-to-end query performance shows promising results and corroborates our assessment as well.

Blacher et al. [5] propose the translation of pure Python code into SQL showing how algorithmic primitives of procedural languages can be mapped to PostgreSQL's declarative syntax. However, the automatic translation of imperative Python code to SQL remains future work. The paper presents simple translations including variables, functions, conditions, loops, and errors –most of which are based on heavy use of WITH syntax. In terms of expensiveness and usability, the translated SQL snippets may be rather complicated for the casual user. In terms of performance, it shows results of running a gradient descent-based logistic regression on NumPy, HyPer [34], and PostgreSQL. Interestingly, HyPer dominates NumPy due to its dynamic tuple-wise parallelization scheme and the ability of the database engine to optimize SQL code as a whole (in a pipelined fashion) and not executing it as individual parts as in NumPy. This result is entirely in sync with our motivation and findings as well.

Another line of work introduces abstractions to transparently define and execute Python UDFs in data processing systems. Grizzly [30] is a front-end framework that exposes a Pandas interface and translates Pandas operations into SQL queries with Python UDFs. AIDA [14] targets mainly linear algebra and provides abstractions for in-database analytics with Python UDFs. It exposes interfaces similar to popular Python statistic packages and translates them into MonetDB Numpy UDFs to support fast linear algebra operations. YeSQL could fit well with such approaches, offering faster UDF execution, richer abstractions given its expressiveness and usability enhancements, and integrations with additional data processing systems due to YeSQL's modularity.

Several prior works target the translation of UDFs to semantically equivalent SQL statements and/or the optimization of UDFs written in SQL using techniques such as compilation, inlining etc. to minimize the context switches between the declarative SQL and the imperative UDF [12, 14, 16, 17, 30, 32, 60, 63, 66]. There is also work on direct embedding of UDFs in native query execution engines [62]. Often such a translation though is not possible or result into deep, complex operator trees, which are hard to execute efficiently [60].

Although these approaches do not relate to Python UDFs, still we find the techniques they use relevant to our work. Froid [60] (a.k.a. MS SQL Server's Scalar UDF Inlining) transforms loop-less T-SQL UDFs to relational algebraic expression and embeds them into SQL. It reports performance deficiencies with T-SQL UDFs and examines solutions tailored for scalar UDFs containing constructs such as variable declaration, select, if/else, and recursive functions. The compiler optimizations it explores include dynamic slicing, constant folding, dead code elimination, and parallelization. Its result advocate huge performance benefits with simple techniques such as 'if a function is simple enough, compile its statements into a SQL subquery that can run inlined into its containing query'. Aggify [28] build on this technology to optimize loops in UDFs. Duta et al. [17] moves into a similar path towards turning interpreted functions into regular subqueries that could be evaluated along with their embracing queries, thus reducing PL/SQL - SQL context switches. It considers a few practical transformations, including compiling iterations into SQL-level recursion. Schüle et al. [63] extends the PostgreSQL grammar to allow lambda expressions and subqueries as table function's arguments and presents a modification of PostgreSQL's JIT compiler framework to inline lambda expressions in table functions. These works identify significant performance concerns with UDFs and move towards solutions to open up user-defined-functions to the SQL executor and optimizer.

UDF fusion has also been explored in various contexts (e.g., loop-fusion [35]). GOLAP [31] fuses external pipelines modifying their source code to eliminate overheads derived from serialization and deserialization steps. Weld [50] translates UDFs into a common intermediate representation (WeldIR) so that they run fused. Intermediate representations (IRs) are also used by HorsePower [11] with HorseIR [10], and MLIR [40]. However, all these techniques support only specific libraries (e.g., matlab, numpy) as they require the libraries to be rewritten in the intermediate representation. In contrast, our work fully supports Python constructs and libraries without any limitation or need to translate the code into an intermediate representation. Still, YeSQL is complementary to these works and could also support their IRs (e.g., through decorator functions).

3 YeSQL OVERVIEW

Python UDFs are commonly used in many application scenarios employing either beefy database servers or minuscule edge devices. A data analyst or a data scientist is more probable to be served by running their complex, resource demanding UDF on a server-based DBMS. On the other hand, several applications in edge computing would benefit from a function-shipping paradigm, which brings the computation closer to a data collector [65]; e.g., a small compute device (e.g., a RaspberryPi) placed in a wind turbine generator to monitor vital metrics and perform data reducing computations. In such scenarios, a UDF running on an embedded system, both in the same process, could be a preferable design choice.

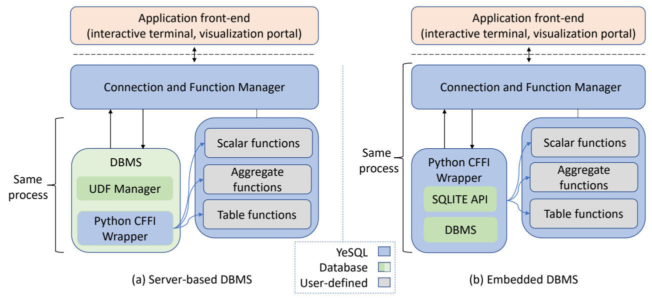


Figure 2: YeSQL: Architecture options

3.1 System Architecture

YeSQL is designed to serve both application scenarios. It can be integrated with either a server-based DBMS (e.g., MonetDB) or an embedded DBMS (via SQLITE API). Figure 2 shows the core components of our architecture on top of both DBMS types.

We distinguish two user roles. Application users (e.g., data analysts, data scientists) submit their queries or workflows to the Application front-end, which in turn propagates them to the Connection and Function Manager. UDF developers create their user-defined functions (gray boxes in Figure 2) and YeSQL registers them in the DBMS. Naturally, the same person may act in either user role.

The Connection and Function Manager (CFM) receives YeSQL queries, transforms them into SQL, and pass them to the DBMS for execution. Note, that using YeSQL syntax is optional. SQL queries using standard SQL syntax simply pass through; still, they benefit from all performance improvements detailed shortly. When integrated with a server-based DBMS, CFM first compiles the UDFs so that they are accessible by the DBMS's UDF manager as an in-process embedded library, and then it submits their declarations directly to the DBMS to run on-demand. When integrated with an embedded DBMS, it submits the UDFs using the Python CFFI wrapper. In this case, the UDFs are executed in the same process with the CFM layer and the DBMS as well.

YeSQL inherits the typical UDF classification into scalar, aggregate, and table functions. The Python CFFI wrapper is the layer that crosses the boundaries between Python and the database engine. With a server-based DBMS, it seamlessly calls the Python UDFs linking the shared library where they are included. With an embedded DBMS, the Python CFFI wrapper submits the UDFs as callback functions and assures the seamless data exchange between the database engine and the Python UDF. SQLITE API natively supports extended-SQL functionality through C UDFs.

3.2 Modularity

YeSQL works as a modular addition to a DBMS, offering easy installation and compatibility with all popular operating systems. Given a data processing system which offers an API to create C UDFs and an ODBC, YeSQL can be integrated and work in synergy with this system with modifications in the Python CFFI wrapper component to meet the specific details of the supported C UDFs (e.g., syntax of `CREATE FUNCTION`, the way that arguments are passed etc.). YeSQL's UDFs run in the same process with the C-UDF API of the DBMS inheriting its characteristics. Thus, if the DBMS runs its native C UDFs out-of-process, then the YeSQL UDFs would also run separately from the query engine. For a server-based DBMS, YeSQL leverages the DBMS's execution model. For example, in MonetDB that has a vectorized execution model, the data is passed via CFFI with one function call as array pointers.

With an embedded database, we exploit SQLITE API's internal streaming architecture and in particular, the Python *generators*, a powerful language pattern that allows co-routines via a `yield` statement. A Python program can be written as if it is in control of iteration (e.g., iterate over an external data source), yet yield values on demand, with control transferred to the database engine for each produced value. In more detail, a generator function returns a lazy iterator, which however does not store its content in memory. This resembles a regular function that uses yield instead of return, and instead of exiting the function after return, the state of the function (e.g., variable bindings, internal stack, instruction pointer, exception handling) is remembered. Yield suspends a function and returns the yielded value to the caller. A generator can resume function execution picking back up right after yield. This mechanism allows putting together data pipelines to handle large datasets efficiently.

Our architecture allows integration of UDFs written in other languages, not only Python, as long as they have a fast foreign function interface and a tracing JIT compiler. For example, UDFs in Lua JIT [41] could also be effectively supported within our architecture.

4 USABILITY AND EXPRESSIVENESS

YeSQL aims at promoting the handling of data-related tasks within an extended relational model. To support diverse data sources, YeSQL operators automatically adapt their schema and data types to the incoming data. YeSQL extends standard SQL with additional syntax and user-defined functions. Currently, all UDFs are written in Python and can use pre-existing Python libraries (e.g., numpy, nltk) via `import`, thus inheriting features that are commonly used by data scientists. This section aims at showcasing the usability and the expressiveness of YeSQL; for space considerations we avoid describing in detail the language constructs. For more details, we refer the interested reader to YeSQL specs [18].

4.1 Functionality

The keystone principle of YeSQL is to enable data scientist develop and run algorithms in SQL seamlessly. Let us consider a simple example. Given a table with all NSF project grant identifiers (these are 7-digit strings), we need to pre-process and text mine the fulltext of a corpus of publications to identify which publications are funded by NSF and create a link to the specific project. Using YeSQL, a simplified implementation could be as follows:

```
select var(`pos`,(select toregex(term) from positives));
select texts.id, projects.id
  from (select id, textwindow(keywords(text),10, 1, 5, '\d{7}')
        from (sample 100 file 'publications.json') as input_pubs)
      as texts, projects
  where texts.middle = projects.grantid and
       regexprmatches($pos, lower(texts.prev||" "||texts.next));
```

The first query uses a table named 'positives'. This table contains terms that are often used by the authors when they acknowledge an NSF project (e.g., "funded by NSF", "this work is supported"). Using an aggregate UDF (`toregex(term)`) these terms are transformed into a regular expression. Function 'var' stores the regular expression in a variable named 'pos'. This variable is stored in Python's execution context and it is accessible by the stateful UDFs.

The second query creates a (virtual) table from a JSON file that stores publications data (`file 'publications.json'`) and samples 100 lines from the file (`sample '100'`). Its returned schema depends on the stored data. In this case, it returns 2 columns: 'id' and 'text'. It processes each text with scalar function keywords (`keywords(text)`), which removes punctuation marks using a precompiled pattern. It runs a sliding window over the text (`textwindow(keywords(text), 10, 1, 5, '\d{7}')`). This function returns three columns and one row per each existence of a seven digit string (NSF project ids) in its input text. The first column 'prev' consists of 10 tokens before the seven digit match, the second column 'middle' contains the match, and the third column 'next' consists of 5 tokens after the match. The result of the above subqueries is joined against the projects table on 'middle' and 'grantid' fields to find occurrences of project grant identifiers in the input texts. Next, the context around the match (columns 'prev' and 'next') is matched against variable 'pos' using regular expression matching to return the grant id occurrences with positive words nearby.

Texts input, sampling, text processing and pattern matching is implemented in Python and executed by a Python VM (CPython or PyPy). Joins and filters are implemented and executed by the DBMS's query execution engine. Note that, with a few additional rules (e.g., terms positioning) to capture corner-cases but with the same easy-to-follow syntax, such a query achieves over 99.5% precision (true positives) in OpenAIRE, a real-world application.

4.1.1 Language features.

The example shows various kinds of UDFs illustrating through them three hallmark features and novel contributions of YeSQL.

Rich support for polymorphic Python UDFs. `file` and `sample` are *polymorphic table functions*; their output is indistinguishable from a regular table as far as the rest of the query is concerned. `textwindow` is a *scalar function*; it runs once per row of the input table, although it may produce multiple rows (and multiple columns, per its output schema). `toregex` is an *aggregate function*; it provides alternatives to standard SQL aggregation, i.e., collapsing multiple rows into one.

UDF fusion. Here, `textwindow` runs directly on the output of function `keywords`, and the same happens with the `sample` and `file` functions. In such cases, YeSQL creates at runtime a new function that fuses the two UDFs in an effort to minimize context switching and data conversion. Moreover, running on a tracing JIT (i.e., PyPy) exposing longer sequences of instructions enables better optimization of the UDF execution itself. Having more than one UDF running in sequence is a common scenario. In this example with text processing there could be many preprocessing steps (e.g., stopword removal, stemming, tokenization, pattern matching, etc.) implemented as UDFs running one after another.

Syntax inversion. YeSQL offers syntactic support for the convenient use and composition of UDFs in a functional language style. The query fragment: "`sample 100 file 'publica tions.json'`" first reads the file containing the publications and then gets a random sample with 100 rows. We elaborate on this shortly in Section 4.2.

4.1.2 Supported UDF types.

YeSQL supports three UDF types, namely scalar functions, aggregate functions, and table functions.

Scalar functions take as input one or more columns from a row and produce one value or a nested table. An example of a YeSQL statement using a scalar function would be:

```
select detectlang('Il en est des livres comme du feu de nos foyers');
>> Results output: french
```

The `detectlang` function detects the language of a text snippet by analyzing its statistical properties. The example snippet is a Voltaire quote, and the correct answer from `detectlang` is that it is in French.

Aggregate functions capture arbitrary aggregation functionality beyond SQL built-ins such as `AVG`, `COUNT`, etc. They produce a single value or a nested table. An example of a YeSQL statement using an external aggregate function would be:

```
select concatterms(a) from
        (select ``term1+term2'' as a UNION
         select ``term2 term3'' as a);
>> Result output : term1+term2 term2 term3
```

The `concatterms` aggregate function concatenates strings of terms together, while keeping the terms disjoint.

Polymorphic Table functions dynamically produce tables that can be used indistinguishably from normal tables in an SQL query. This yields significant flexibility and expressive power. With polymorphic table functions, YeSQL can integrate many heterogeneous data sources in a query. Example sources of table datasets are files, SQL query resultsets, or even the output of external programs. Table functions can be used in a variety of ways. The simplest is as a *parametric table*. For example, in the following query `file` reads the tab-separated file "`continents.tsv`" as a table.

```
select * from file('./demo/continents.tsv');
```

4.2 Syntactic Inversion

Elevating user-defined functions to first-class citizens inside a relational query requires syntactic support. YeSQL offers support for table function chaining. We call this feature *syntax inversion.*

Let us consider an example UDF integration into an existing DBMS. The following PostgreSQL query uses three UDFs: (a) `xmlparse`, to parse an XML data source and return text rows, (b) `rowidvt`, to add a row-id column to the resulting table, and (c) `sample`, to produce a random subset of the input rows.

```
select * from
 sample(10000, 'select * from
        rowidvt(''select * from
                 xmlparse(''''select xml from table'''')'')');
```

This syntax is quite cumbersome. The need for complex quoting alone is a strong deterrent from practical use of chained UDFs. YeSQL offers an alternative to writing a nested subquery for each table returning UDF and using syntax inversion allows us to express the query as follows:

```
sample 10000 rowidvt xmlparse select xml from table;
```

Hence, YeSQL allows table UDFs to be written as a space-delimited sequence, denoting that each UDF is passed as a parameter to the UDF on its intermediate left. The syntax of a table function (e.g., `myUDF`) would be: "`myUDF 'p1' 'p2' ... np1:val_np1 ... SQL_query`". The UDF accepts (optional) positional parameters (`p`), named parameters (`np`) with their values (`val_np`), and is applied to the result of a SQL query. At the translation, CFM converts this syntax into:

```
select * from myUDF('p1','p2',...,'np1:val_np1',...,'query:SQL_query');
```

Type declaration. Some DBMSs require an up-front schema declaration; i.e., each UDF needs to declare a single-typed schema for its result (e.g., [54]). YeSQL UDFs do not need to declare a schema for output data. Their return type is dynamically determined by the structure of the data. Thus, UDFs are naturally polymorphic; a UDF may have different output types (e.g., text vs. arithmetic data) when it is used over different data. For example, the YeSQL UDF *jpath* parses JSON input and produces text or arithmetic output depending on what is contained in the JSON text input. The output data types of *jpath* do not need to be known in advance or declared in any way. For SQLITE API, this is achieved with using virtual table interface and each polymorphic function is internally implemented using a virtual table. For a DBMS such as MonetDB, polymorphic functions are implemented using Python loader functions to load data into an in-memory temporary table without previously defining their output columns and types.

4.3 Code Generation

Support of YeSQL language takes place at the CFM layer. Currently, this layer is written in Python. CFM consists of the following components: (a) function manager, (b) parser, and (c) code generator.

4.3.1 Function manager. It submits UDFs to the DBMS. The UDFs are categorized according to their types (i.e., scalar, aggregates or polymorphic tables) and are placed into Python files inside YeSQL's UDF directory. The function manager accesses the packages which contain the UDF definitions and submits them into the DBMS. Submission of the UDF is implemented differently, depending on whether YeSQL is integrated with an embedded or a server DBMS. In the first case, the submission is done via SQLITE API functions [69]; i.e., a Python UDF is wrapped using CFFI and then submitted as a function callback using the `sqlite3_create_function` function [8]. When running on a server-based DBMS, the submission of a UDF is done using SQL `CREATE FUNCTION` statements. For example, in MonetDB we use the existing C/C++ UDFs [47]; i.e., a Python UDF is JIT-embedded into a C UDF using CFFI functions and then submitted using a `CREATE FUNCTION` statement. When the C code is invoked, it calls a function pointer which in turn calls the Python function [9]. We work similarly for any DBMS supporting C UDFs.

4.3.2 Parser. It parses the YeSQL language before submitting the query to the DBMS. If the query does not involve any YeSQL UDFs the parser submits the query as is. Otherwise, (a) it converts subqueries written with YeSQL's inverted syntax to standard SQL format, and (b) it propagates polymorphic UDFs to the code generator that converts them to the appropriate SQL code.

4.3.3 Code generator. Its core functionality is to support polymorphic UDFs. The code generation is engine-specific depending on the underlying DBMS. For example, UDFs that return polymorphic schema are implemented with loader functions in MonetDB and with virtual tables when integrated with the SQLITE API. Hence, the example polymorphic UDF: `select * from file('data.csv'');`
in MonetDB is converted into:

```
create temp table temp_name from loader file('data.csv') on commit drop;
select * from temp_name;
```

and with the SQLITE API would be:

```
create virtual table if not exists
   temp.vt_name using file('data.csv','automatic_vtable:1');
select * from temp.vt_name;
drop table if exists temp.vt_name;
```

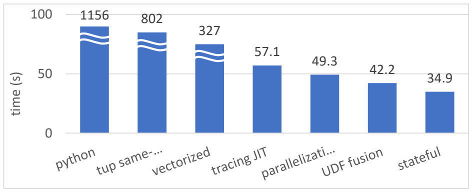


Figure 3: Largest factors in boosting Python UDF execution

Note, that in both cases the intermediate results are not materialized on the disk. In MonetDB, the intermediate result is stored in a memory resident, temporary table. With the SQLITE API, intermediate results are defined as a *lazily evaluated* virtual table. A query containing more than one UDF is converted into multiple subqueries each with their own temp/virtual tables, which are executed recursively. When the subqueries terminate, the corresponding temp/virtual tables (by default) are dropped. Loader functions and virtual tables are used to support polymorphic table UDFs. For table UDFs with static returned schema, YeSQL follows a faster path using the table functions that most DBMSs support. Implementing polymorphic UDFs without intermediate queries is interesting future work, which would enable optimizing the whole workflow at once. One approach would be to JIT-create the wrapper function according to the current query and data types.

5 PERFORMANCE ENHANCEMENTS

The performance enhancements in YeSQL aim at avoiding the impedance mismatch between the relational (SQL) evaluation and the procedural (Python) execution. This mismatch causes two major overheads. (a) Context switching overhead: one facility needs to invoke the other through various levels of indirection. This is potentially expensive when performed frequently. (b) Data conversion overhead: data is represented differently in the two environments and need to be wrapped/unwrapped or checked (e.g., for overflow) and encoded/decoded. To remove these overheads, we employ five techniques: tracing JIT compilation, seamless integration with the DBMS, UDF fusion, parallelism, and support for stateful UDFs.

5.1 Where the Time Goes

To put things in a perspective (and to answer why these techniques), we discuss a micro-benchmark showcasing the additional overheads incurred by this mismatch. Let us consider a query with four UDFs:

```
select udf1(postal_code), udf2(facts_and_features), udf3(udf4(url))
from zillow;
```

These are three functions that extract the `postal_code`, `bds`, and `id` from the Zillow dataset, and `udf4` is a string manipulation function converting `url` to lower case. We run the query first as a spawned CPython process as tuple-at-a-time. Next, to remove these overheads, we try the boosting techniques one at a time, as follows: (a) vectorized execution using embedded NumPy, (b) tracing JIT-compilation with PyPY, (d) employ parallelism with multi-threaded execution, (e) UDF fusion on JIT, and (f) stateful UDF execution. Figure 3 presents a performance breakdown illustrating the extent that each technique contributes to query performance.

Although any of these techniques in isolation helps boosting UDF execution, applying them all and in a specific order increases the optimization opportunities and improves performance by a factor of 33x starting from a spawned, tuple-at-a-time CPython process, which resembles an out-of-the-box execution on PostgreSQL with PL/Python (our experience shows that many data scientists do start here, either on PostgreSQL or on Spark with PySpark, etc.) and 10x from an embedded vectorized execution, which will be a better starting point should someone pick a tool with improved support for Python UDFs (e.g., MonetDB, DuckDB [15]). Some of these techniques are orthogonal to each other, whereas others such as parallelization need to be performed in the right order; e.g., applying parallelism before JIT would not have a significant overall effect as we see in Section 6.3.4. Similarly, applying fusion before JIT and parallelization would miss the chance to execute the fused components on a single trace and in parallel. And applying JIT before vectorization will have less impact as then tracing would operate on a single tuple, instead of tracing a batch of a tuples in vectorized execution (see also Section 6.3).

5.2 Tracing Just-In-Time

JIT compilation boost performance of programs by compiling parts of a program to machine code at runtime. In contrast to method-based JIT compilers that translate one method at a time, tracing JIT uses frequently executed loops ("hot loops") as their unit of compilation. This has an excellent fit to UDFs, as they execute frequent complex calculations iteratively through the tuples of a table. Thus, in theory, they can benefit from a tracing JIT compiler.

We employ the PyPy dynamic compiler (a.k.a. JIT compiler), a high-performing engine for Python program execution. The YeSQL query compilation meshes well with PyPy compilation and can be viewed as a pre-optimization step, in much the same way as loop unrolling or inlining enable a multitude of other optimizations in a traditional static compiler. By fusing UDFs and exposing larger chunks of Python code, YeSQL allows PyPy to perform better compilation. This is especially the case since PyPy is a trace-compiler: it produces optimized machine code at the level of traces, i.e., series of instructions commonly executed in sequence[4]. Thus, by exposing longer sequences of instructions, especially for relatively simple UDFs, we enable more optimization in the Python execution itself.

PyPy has some extra advantages: (a) it supports the standard Python library [57] and most of the popular packages [56], whereas new packages are regularly added as it is an active project; (b) it optimizes already existing dynamically typed Python code and packages without any editing, and does not require a-priori compilation; (c) it supports exception handling same as in CPython, thus in YeSQL UDFs exceptions are handled inside the UDF wrapper; and (d) it offers CFFI, a foreign function interface to efficiently interact with C code. Moreover, CFFI supports CPython as well allowing the use of CPython interpreter for UDFs that import (currently) not supported packages e.g., scikit-learn. This is not a limitation, as PyPy optimizes Python code and not supported packages are packages mainly implemented in C and thus, are also fast in CPython.

[4] The sequential execution of a translated trace is guarded dynamically. A guard is checked at every branch instruction. If execution diverges from the trace, the compiled code branches to an interpreter (or to a different compiled trace, if one exists and if overlapping traces are supported) and continues executing the appropriate off-trace instructions in interpreted mode.

PyPy also facilitates the integration with a DBMS. Let us consider for example the case of MonetDB. CFFI's internal array representation is compatible with Numpy, which fits well with MonetDB's Python support [59]. In our implementation, such an array is seamlessly passed to CPython UDFs as a numpy array using the `numpy.frombuffer` function that interprets a buffer as an one-dimensional array *without data copies*. When running on MonetDB that supports vectorized UDFs, the pointer to the whole column is passed with *one function call*, minimizing multiple foreign function calls overheads. Moreover, since PyPy enables its own vectorization these conversions are transparent to the UDF developer [52].

The UDF that the user writes runs per tuple but it is optimized by the tracing JIT. The example below shows the low level implementation of a scalar UDF in MonetDB that counts string length:

```
@ffi.def_extern()
def lenstr_wrapped(input,insize,result):
    for i in range(insize):
        result[i] = lenstr(ffi.string(input[i]))
    return 1

def lenstr(val):
    return len(val)
```

The `lenstr_wrapped` function is embedded (i.e., decorated with `ffi.def_extern()`) and called by the DBMS. Under the hood, we obtain a cdata pointer-to-function object, which can be passed to C code and then works like a callback: when the C code calls this function pointer, the Python function is called [9]. This is also a wrapper that makes the appropriate conversions using CFFI before and/or after calling the UDF. The results are assigned to the preallocated result array which is a cdata object. The `lenstr` function is the UDF that the UDF developer actually implements.

The CFFI wrapper works a little differently with an embedded DBMS. It submits the UDF as a function callback with the appropriate conversions and the function is called by the database.

5.3 Seamless Integration with DBMS

YeSQL supports seamless integration with the DBMS. For doing so, UDFs are wrapped using embedded CFFI [7]. During the execution of a UDF, data is transferred to CFFI as pointers to cdata objects without any data copies. Integers and float columns are used directly by Python. For string columns, we have three options: (a) `ffi.string`, which transforms the string to a format that is understandable by PyPy (or CPython), (b) `ffi.buffer`, which returns a Python `memoryview` without copying the string; `memoryview` can be seen in Python as an array of characters and is also supported by several Python modules including the regular expression package, and (c) `direct pass`, passing directly the pointer to the C string enables low level optimizations manipulating the pointer in a C manner.

5.4 UDF Fusion

Producing small and reusable UDFs is a common practice as it enhances productivity. For example in text mining, a UDF which tokenizes the input text can be reusable in many different workflows. According to the workflow it is usual to mix multiple UDFs together; e.g., tokenization, stemming, stopwords removal, and other normalization algorithms are called one after the other.

A performance enhancement comes with UDF fusion. When more than one UDF run in sequence and can be fused, the fusion takes place at the level of the CFFI wrapper function. A new wrapper function is created just-in-time and pipelines the UDFs. This has two benefits: (a) the CFFI conversions are eliminated, and (b) since the UDFs are called by the DBMS they run on a different trace. By fusing them, we expose longer sequences of instructions so more optimization is enabled by the tracing JIT.

When two UDFs are fusable (i.e., the second UDF's input data is the same as the first UDF's output and the argument data types are available in the query plan), the CFFI wrapper creates a new UDF pipelining the two UDFs involved. Let us consider these two UDFs:

```
def multiply(input):
   return len(input)

def add(input):
   return add+1
```

If they are fusable according to the query plan of a specific query, the CFFI wrapper creates a wrapper UDF like the following:

```
@ffi.def_extern()
def fused_add_multiply(input, count, result):
   for i in range(count):
      val = ffi.string(input[i])
      result[i] = add(multiply(val))
   return 1
```

Hence, with fusion: (a) The conversions from/to C data objects (e.g., with the `ffi.string` function) run once to provide the input to the first UDF of the sequence, and once at the end to return the final result. Without fusion, the DBMS would call two UDFs incurring unnecessary conversions and context switches. (b) We provide longer traces to the tracing JIT and also achieve loop fusion [35]; both UDFs are called in the same loop, avoiding the need for two loops, one per UDF.

Similarly fusion of aggregate and table UDFs that run in a pipeline, still eliminates context switches between the DBMS and the UDF language and produces larger traces. Loop fusion is a little more complicated in this case. For example, we also achieve loop fusion with a scalar function running before an aggregate by defining the aggregate UDFs as a Python class with 3 functions (`init`, `aggregate`, and `finalize`). This is not new, several existing systems are modelling aggregate UDFs in a similar way [67, 70, 75]. Thus, an aggregate UDF that calculates average is written as follows:

```
class Average:
    def __init__(self):
        self.count = 0
        self.sum = 0

    def aggregate(self, val):
        self.count+=1
        self.sum += val

    def finalize(self):
        return self.sum/self.count
```

If this UDF can be fused with an upstream scalar UDF, the JIT wrapper function would combine both UDFs in the same loop:

```
@ffi.def_extern()
def fused_average_scalar(input, count, result):
    avg = Average()
    avg.init()
    for i in range(count):
        avg.aggregate(scalar(input[i]))
    result = avg.finalize()
    return 1
```

5.5 Parallelism

In most cases, parallelism improves the performance and scalability of a program. Most DBMSs exploit multi-threaded execution to run independent operators. However, when these operators are Python UDFs their performance when run in parallel is limited by the Python Global Interpreter Lock (a.k.a. GIL). GIL is a mutex (or a lock) allowing only a single thread to hold the control of the Python interpreter and thus, only a single thread is allowed to be executed at any given time. GIL prevents deadlocks and by itself it does not add too much overhead. Still, it causes a significant performance bottleneck in CPU-bound and multi-threaded code.

GIL is also enabled during the creation of a Python Object (i.e., this happens when the database data are translated to be used by a Python UDF). GIL is released when a C function is called and then acquired to create the Python object. Releasing and acquiring GIL is also a costly operation. While transforming a data input to be used by a Python UDF, GIL takes place once per input value. However, in PyPy, the creation of a Python object is faster as less memory is required to create and store a PyObject. Moreover, PyPy allows for optimized releasing and acquiring GIL. In CFFI, GIL is released lazily, i.e., the thread doing the call to C just marks GIL as released by setting a global variable, with no synchronization [25].

5.6 Stateful UDFs

Most data processing systems supporting Python UDFs support stateless UDFs, i.e., only their output lives beyond the end of the UDF. For example, Tuplex [68] uses a slab allocator to provision memory from a thread-local, pre-allocated region for new variables within the UDF, and frees the entire region after the UDF returns and Tuplex has copied the result. Still, stateful UDFs open up many interesting opportunities for algorithm development in areas such as data analysis and data science. Besides this, stateful UDFs may enable some specific performance optimizations as well, e.g., pre-compiling a pattern instead of compiling it once per each row.

When running UDFs on an embedded DBMS the UDFs are by default stateful, i.e., they can access states defined outside their code definition. When running on a server based DBMS such as MonetDB, the state is also available during the processing of different rows since the whole column is passed to the CFFI wrapper. Moreover, the state is also shared across different UDFs. The UDF developer provides her UDF as a python module with functions; i.e., it may import and setup packages in the global layer. The Function Manager creates a CFFI embedded function that wraps these functions and at run time, it calls them from the pre-imported module. This is important for performance, as the developer may run costly operations (e.g., using nltk data downloaders or assign pre-compiled regex patterns) once at global scope. For example, the following UDFs are equivalent but the left one imports the package and pre-compiles the pattern at global scope.

```
import re                               def returnint(val):
getint = re.compile(r'(\d+)')             import re
def returnint(val):                       match = search(r'(\d+)',val)
 match = getint.search(val)               if match:
  if match:                                 result[i] = int(match.group(0))
   result[i] = int(match.group(0))        else:
  else:                                     result[i] = 0
   result[i] = 0
```

6 EVALUATION

The YeSQL codebase is ~66K lines of Python and C++, including ~18.5K lines for the code definitions of 150+ Python UDFs currently supported. We evaluate YeSQL with three representative, data science pipelines and micro-benchmarks of specific design features.

6.1 Experimental Setup

6.1.1 Hardware and Software. We run all experiments on an Intel Core (Ivy Bridge E) i7-4930K processor with 3.40GHz and 6 cores / 12 CPUs. The server has 64GB main memory and runs Ubuntu 20.04. Unless otherwise stated, we executed all measurements with cold caches on SSD disks, and report the average of 5 executions. We compared YeSQL against Tuplex [68], MonetDB (v.11.41.11), PostgreSQL (v.12.9), the latest version of a commercial distributed analytic database engine (dbX), Pandas (v.1.3.5) and Spark (PySpark, v.2.4.7). In the experiments we also used Cython (v.0.29.25), Numba (v.0.54.1), Nuitka (v.0.6.19.1) and Compiled_UDF_engine [13]. YeSQL's extensions to MonetDB and SQLite employed PyPY (v.7.3.6 with GCC 7.3.1), CFFI (v.1.14.6), CPython (v.3.8.10), and SQLite (v.3.31.11).

6.1.2 Datasets. We measured YeSQL's end-to-end pipeline performance using three pipelines: *zillow*, *flights* and *text-mining*.

The zillow and flights pipelines are obtained from Tuplex's github repository and they are the same used in the Tuplex paper [68]. The zillow data snapshot we use contains listings from the Boston, MA area. It contains 10 columns and three size variations: 1GB/5.6M rows, 5GB/28.6M rows, and 10GB/56M rows. The flights data originated from a Trifacta tutorial [27] and the Tuplex team extended it by joining it with airport and airline data from other sources [51, 73]. It contains 110 columns and we use three size variations: 1.6GB/5M rows, 3.2GB/10M rows, and 6.4GB/20M rows.

The text mining pipeline comes from a real-world application, OpenAIRE. It runs on the plain texts of open access research publications and its purpose is to produce links between publications and projects. The workflow involves two pre-processing operators, i.e., `tokenization` and `stopwords removal`, and a `pattern extraction` operator implemented as UDFs. The result of the `pattern extraction` is `joined` with a database table. Finally, a `pattern matching` operator runs on the join results to filter out false positives.

6.2 End-to-end Pipelines

6.2.1 Zillow. Figure 4 presents the results of the zillow pipeline on different systems, and varying data sizes and degree of parallelism. This pipeline involves 8 compute operators and 3 filters.

YeSQL with tracing JIT on MonetDB (`mdb.pypy`) outperforms the other candidates in both single-threaded and multi-threaded executions. MonetDB with CPython (`mdb.python`) suffers from slow parallelism due to GIL that is released and acquired each time a Python UDF is involved. `mdb.pypy` has also GIL but it benefits from multi-threaded execution as GIL is released lazily during CFFI conversions (see also 5.5). In-process UDF execution enables faster execution than Spark/PySpark (`spark`), which spawns a main Java process and separate Python processes. Our implementation differs from Tuplex (`tuplex`) in terms of parallelism. Tuplex achieves parallelism via data partitioning, which adds an extra overhead to create the partitions and merge back the result. MonetDB parallelizes the query operators without necessarily requiring the data to be partitioned. Overall, YeSQL is on average 45% and 28% faster than tuplex, in multi-threaded and single-threaded execution, respectively.

SQLite (`sqlt`) supports only single-threaded execution; still, it remains competitive in this pipeline as the UDFs run in the same process with the DBMS. However, due to its tuple-at-a-time execution model it does not support vectorization and this slows down its execution for two reasons: (a) the Python UDF calls are made once per tuple, which adds an extra overhead and especially here that we have foreign function calls, and (b) it provides short traces (just 1 tuple) to the tracing JIT. MonetDB provides the whole vector to the tracing JIT, so it enables better optimization. Hence, tracing JIT speeds up MonetDB's multi-threaded execution by 10x (single-threaded by 6x), and SQLite only by 3x.

6.2.2 Flights. Figure 5 presents the results of the flight pipeline on different systems, and varying data sizes and degree of parallelism. This dataset differs from zillow, as it contains a large number of columns (110), two small tables used in joins, and more operators, 23 operators on the larger table, 3 joins, and 1 filter.

The results reveal very similar trends with the zillow experiments. YeSQL with tracing JIT on MonetDB (`mdb.pypy`) outperforms the other candidates. MonetDB with CPython (`mdb.python`) does not perform well in multi-threaded execution, as there are much more UDFs running in parallel and the inter-communication overheads are significant. Tracing JIT seems to be fixing this issue at a reasonable extent. Here, Tuplex (`tuplex`) performance is on par with YeSQL in single-threaded execution. The reason is that this pipeline involves 6 UDFs running on the largest table and return strings. Tuplex is an end-to-end JIT engine and the entire flow runs in the same execution context. However, MonetDB interns its strings using a hash dictionary, which hurts single-threaded UDF execution. String interning is a useful optimization feature for filters/joins on string columns but such operations do not exist in this pipeline. YeSQL with JIT outperforms tuplex in multi-threaded execution (on average 53% faster), for the same reasons we explained earlier. The results of Spark and SQLite resemble a lot our findings in the zillow experiments, which strengthens our observations.

6.2.3 Text mining. This is a real-world pipeline with the characteristic that the tuples involved contain the fulltext of publications. We selected this pipeline to access the performance of heavy UDFs running on smaller tables but with larger values.

Figure 6 presents the results of this pipeline with varying data sizes and with various implementations, i.e., CPython, Numpy, and PyPy JIT on a server-based engine, MonetDB, and an embedded one, SQLite. A key finding in this experiment is that vectorization in Python UDFs does not makes any difference in this use case. Instead, SQLite implementations with CPython run similarly and a little faster than MonetDB. Note, that in this experiment we used an index in SQLite to help with the join; MonetDB does not create a physical secondary index but decides internally which column search accelerator(s) to create and use [46]. Having smaller columns means that there are not many foreign function calls per UDF execution, thus the function calls overhead is negligible and largely dominated by the overhead of the heavy python UDFs. On the other hand, the tracing JIT works better with vectorized execution, which confirms our claim that providing larger traces

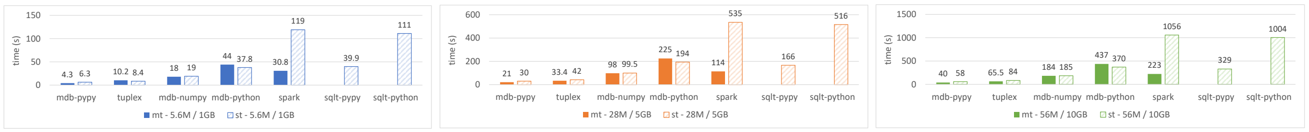


Figure 4: Zillow pipeline for varying data sizes and parallelization

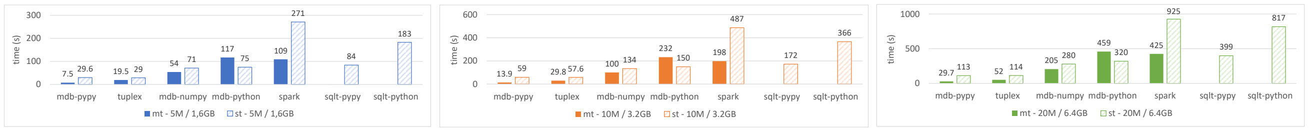


Figure 5: Flights pipeline for varying data sizes and parallelization

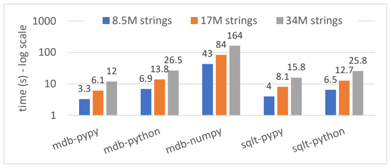


Figure 6: Text mining pipeline (time in log scale)

to the tracing JIT enables more optimization, which in YeSQL is achieved via vectorization and UDF fusion.

Finally, with YeSQL's extension to MonetDB to support tracing JIT, we achieve significantly faster execution than MonetDB's standard Numpy UDFs for 2 reasons: (a) Numpy UDFs in MonetDB do not allow parallelism and this is illustrated in the execution times, and (b) for string parameters, MonetDB transforms the C strings into Python Objects (CPython) and loads a Numpy array with those objects. This is heavier than in the tracing JIT Python's compiler, which uses utf-8 internally for unicode strings [58] as MonetDB also represents its strings with utf-8 [45].

6.3 Scrutinizing YeSQL

Here, we present a series of micro-experiments to delve into implementation details that impact the performance of YeSQL. Unless otherwise stated, in these experiments we use the flights pipeline.

6.3.1 Tracing JIT. First, we investigate the effect of tracing JIT (PyPy in YeSQL) into the UDF execution. To evaluate the performance of PyPy in UDF execution we run an experiment with a UDF that splits a string and compute the average word length on TPC-H [71] `partsupp` table at SF-10 (i.e., this is the exact same microbenchmark presented in [37]). We compare the UDFs running on YeSQL's implementation with PyPy and CPython on MonetDB, against the native Python, Cython, Nuitka, and Numba implementations obtained by [37]. Figure 7a shows that YeSQL's implementation with PyPy as tracing JIT and CFFI to interact with C, UDF execution outperforms the other alternatives. Specifically, using the tracing JIT the UDF runs 3x faster than our CPython implementation, while Cython is able to speedup the UDFs by 2x compared to CPython. Tracing JIT UDFs in YeSQL run 1.7x faster than Cython. Interestingly, the other alternative transpiler (Nuitka) and compiler (Numba) options do not seem to perform better. Although they speed up CPython interpreted UDFs by 2.4x and 1.9x, they are still much slower than YeSQL. But more importantly, these approaches suffer from high compilation times, which in combination with the significant limitations we discussed in Section 2 for Cython, Nuitka and Numba, renders them unsuitable for UDF execution in DBMS.

6.3.2 Seamless integration with a DBMS. We examine various aspects of the integration, including performance affecting designs such as UDF execution vs. vanilla SQL and string transfer alternatives, but also performance across various data processing engines.

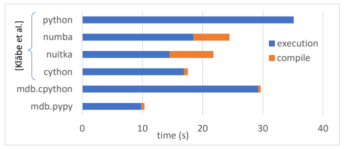
Experiment 1. To illustrate the benefits of seamless integration, we run on MonetDB an experiment with the following SQL query and its incarnation as a Python UDF.

```
[SQL:] select case
                when depdelay='' then 0
                else cast(cast (depdelay as float) as int)
             end as airtime
       from flights;

[UDF:] @ffi.def_extern()
       def toint_wrapped(input,count,result):
          for i in range(count):
             result[i] = toint(ffi.string(input[i]))
          return 1

       def toint(val):
           return int(float(x)) if x else 0

       select toint(depdelay) as depdelay from flights;
```

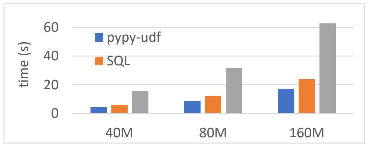
Running on 80M rows from the `flights` dataset, the SQL query runs in 12.6 sec and the PyPy UDF runs in 8.7 sec. This is an interesting result showing that UDFs appropriately orchestrated using tracing JIT, a CFFI wrapper, and the seamless integration design, when running on a DBMS may achieve similar or even better performance with the native SQL queries, which presumably constitute an optimal implementation for DBMS. This result raises a question about whether we need to translate UDF code to an intermediate representation (see Section 2 for a discussion about works converting UDFs to IR) if we can match optimal performance with careful optimization and tuning. Numpy UDFs are much slower in this query (31.6 sec) as expected, since they spend cycles to convert strings to PyObjects to fill the Numpy array. Figure 7b presents results of this experiment with data sizes span 40M, 80M, and 160M rows. The trends we described are consistent in all three cases.

Experiment 2. We evaluate the overhead of transferring string columns using `ffi.string` compared to `direct pass` (i.e., processing directly on the c data object). The following code gets a string column and extracts the part of the string after the comma. The text snippet ‘input[i]+j+1‘ moves the pointer j+1 positions.

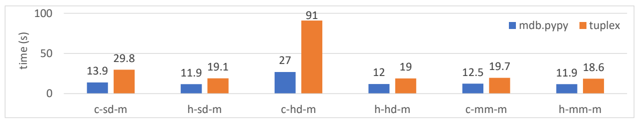
```
@ffi.def_extern()
def parsestr_wrapped(input,insize,result):
   for i in range(insize):
     result[i] = parsevalue(input[i])
   return 1
def parsevalue(val):
     j = 0
     while True:
```



(a) Comparison of a single UDF



(b) Seamless integration



(c) Cold cache vs. Hot cache on SDD/HDD/tmpfs (multi-threaded)

Figure 7: Evaluating the performance of a single UDF, seamless integration, and various setups

```
        if val[j] == ',':
            break
        j+=1
    return val+j+1
```

The following UDF achieves the same functionality by transforming the input to Python strings.

```
@ffi.def_extern()
def parsestr_wrapped(input, insize, result):
    tmpstrs = [parsevalue(ffi.string(input[i]).decode()).encode()
            for i in range(insize)]
    for i in range(insize):
        result[i] = ffi.from_buffer(memoryview(tmpstrs[i]))
    return 1

def parsevalue(val):
    return val[val.rfind(',')+1:]
```

Running on a column from the flights pipeline (average value length: 13) with varying data sizes, the first UDF is faster on average by 1.9x due to the applied data transformations. For example, for 10M rows the first UDF runs in 1 sec and the second in 2 sec. These executions times include the following MonetDB specific operations: (a) the returned strings are first inserted in a malloced intermediate result array; and (b) the strings are then copied and inserted into a hash dictionary before the result array is freed.

Experiment 3. We examine the impact of various setups: cold (c) vs. hot (h) caches; SSD (sd) vs. HDD (hd) vs. shared memory (tmpfs) (mm); with parallelization (m) or not. In this experiment, we compare the fastest implementations of YeSQL and tuplex on the flights pipeline (10M rows). Our findings are consistent with the results in Section 6.2. Specifically, tuplex performance is on par with YeSQL in single-threaded execution. However, in multi-threaded execution YeSQL is faster in all cases at a factor that varies between 1.56x to 3.37x, according to the setup. Tuplex is highly affected by the execution setup (i.e., its execution times varies from 18.6 sec to 91 sec), and achieves its best performance with hot caches. On the contrary, YeSQL's implementation on a columnar database engine runs efficiently in all setups, with times varying between 11.9 sec to 27 sec. Figure 7c shows the results of the various combinations.

Experiment 4. Concluding the micro-experiments on the integration with a DBMS, we revisit the comparison presented in Figure 1. This experiment evaluates a single UDF on popular DBMSes that support Python UDFs: PostgreSQL with PL/Python, MonetDB with NumPy, SQLite, a popular commercial parallel and distributed column-store DBMS (denoted dbX), Spark with PySpark, Pandas, Tuplex, and YeSQL extensions to MonetDB and SQLite with CPython and PyPy. The query uses real estate data from zillow, returns 7M single-column rows with descriptions of apartments (example value: '2 bds, 1.0 ba, 856 sqft'), and includes a scalar Python UDF that extracts an integer representing the number of bedrooms. YeSQL's extensions to MonetDB with either CPython or PyPy outperform all alternatives. PostgreSQL, Spark, and dbX run in CPython's interpreter as a separate process involving also inter-process communication overheads, and so, they run 68x, 45x, and 38x, respectively, slower than YeSQL. Pandas is 10x slower than YeSQL, as Pandas is generally optimized for numeric operations and not for strings [72]. The performance of Tuplex (6x slower) and Numpy (8x slower) follows here as well the trends explained in the previous experiments. YeSQL extensions to MonetDB are faster than those on SQLite due to the vectorized execution model; the data is passed to the UDF through one function call whereas SQLite calls the foreign function once per tuple. Notably, YeSQL on SQLite as an embedded database with PyPy (mostly) and CPython remains competitive, showing that YeSQL, besides its application alongside server-based DBMSs, is also an excellent implementation for lightweight architectures and for a broad variety of applications (e.g., edge-computing).

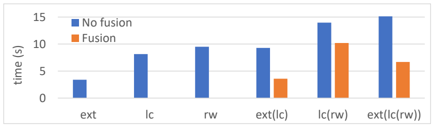


Figure 8: UDF fusion

6.3.3 *UDF fusion.*

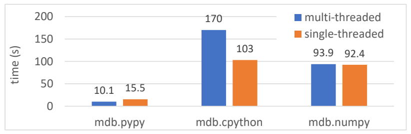
We run this experiment using 10 million text snippets from the text mining dataset. Figure 8 shows the execution times for the following query (and also different combinations of the involved UDFs) which runs with and without UDF fusion:

```
select extractnumber(lower(remove_small_words(text))) from texts;
```

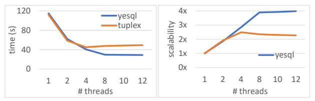
This query first removes the words with less than three characters (`rw`), then converts the input to lower case (`lc`), and at last extracts an integer number using regular expressions (`ext`). As shown, in all cases fusion enables more optimization, and when fusing all three UDFs the query executes 2.26x faster. With UDF fusion the CFFI conversions and the context switches between the DBMS and the UDF are eliminated. However, fusion may have different performance gains according to the characteristics of the UDFs. Specifically, when the fused UDF has string output and can be fused the gain is larger. For example, the execution time of fuzed `ext` and `lc` is 3.6s, while the execution time of `lc` alone is 8.1s. This happens because MonetDB interns its strings using a hash dictionary that adds an extra overhead, which however is optimized out as the UDF is fused with one that returns a numeric value.

6.3.4 *Parallelism.*

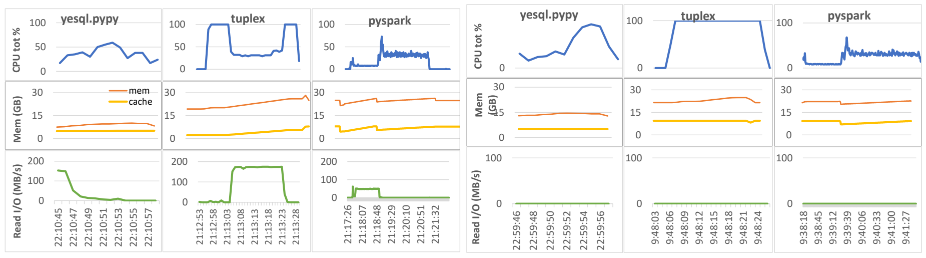
We experiment with the execution times of 9 UDFs from the flights pipeline using our implementation of PyPy and CPython UDFs, and also MonetDB's default Numpy UDFs. We run the UDFs in single-threaded and multi-threaded executions. Figure 9a shows that Python implementations are faster when no threads are used due to the limitations derived from releasing and acquiring GIL (see also Section 5.5). Vectorized MonetDB's Numpy



(a) Multi-threaded vs. single-threaded



(b) Scaling degree of parallelism



(c) Resource usage: YeSQL, Tuplex, PySpark with cold and hot caches

Figure 9: Evaluating UDF fusion, parallelism, and resource utilization

UDFs also do not get any significant speedup from multi-threaded execution. However, when running on PyPy we reap benefits from multi-threaded execution running 50% faster than a single thread.

6.3.5 Stateful UDFs. We investigate the effect of stateful UDFs by running a pattern extraction UDF on MonetDB, with both a stateful and a stateless implementations, on 56 million string values from the zillow pipeline (average length: 24 characters). The first, stateful UDF executes in 7 sec while the second requires 14.6 sec. The speed up is explained as follows. The stateful UDF enables precompiling the pattern using `re.compile` at global scope, while the stateless UDF runs the pattern extraction per tuple without using any precompiled pattern.

6.3.6 Scalability. We evaluate the scalability of YeSQL and tuplex, the two most scalable solutions in our analysis, using the flights data (20M). We try 1 to 12 execution threads (the maximum allowed in our setup). Figure 9b shows that YeSQL achieves a speedup up to 4x for 8 threads, which is the sweet spot of our server. Tuplex follows with a speedup up to 2.4 but it reaches a plateau at 4 threads. YeSQL incurs a lower performance degeneration 7% to 50% (tuplex goes up to 81%), which renders it a good fit for multi-core environments.

6.3.7 Resource Usage. We profile resource usage (CPU, memory, and disk) of YeSQL with PyPy, Tuplex, and PySpark. We monitor metrics such as total CPU (%), memory/cache (GB's), and disk read i/o (MB/s). We run an experiment with the flight dataset (10M rows), multi-threaded execution, and cold/hot caches (see Figure 9). While running on cold caches, YeSQL finishes in 12 sec, with fast loading in 3 sec, moderate use of memory and CPU. Interestingly, YeSQL starts processing before loading finishes. Tuplex finishes in 36 sec, with slow loading ($\tilde{1}$8 sec), aggressive use of memory and cache, and excellent CPU usage while not loading. PySpark finishes in 246 sec, with very slow loading ($\tilde{8}$2 sec), aggressive use of memory and cache, and moderate CPU usage when loading completes. On hot caches, YeSQL finishes in 10 sec with moderate use of memory. However, it starts with moderate CPU usage (most UDFs run at the beginning and they are affected by GIL), and continues with high CPU usage when the joins kick in. Tuplex is improved with hot cache. It finishes in 21 sec, still with high memory and cache usage, and fully utilized CPU. PySpark finishes in 189 sec, with high memory and cache usage (similar to Tuplex), and moderate CPU utilization. Note that YeSQL outperforms Tuplex and Spark while using significantly less memory. YeSQL's implementation on a columnar DBMS helps loading and processing only the required columns avoiding unnecessary overheads.

6.4 YeSQL in Practice

The usability and expressiveness of our YeSQL language was put to the test with an assignment given to 380 undergrad students, with little experience with SQL. They were asked to develop two algorithms: (a) Document similarity with TF-IDF and (b) Document classification using a preexisting training set with weighted terms, in two implementations: (a) in YeSQL and (b) in Python and SQL but without UDFs, and report on their experience. 328 (86.3%) students completed successfully the task and 165 (43.4%) scored an excellent grade. The success rate was higher than past years, when the assignment did not include YeSQL. Most students found programming with YeSQL easy and liked that their YeSQL code was smaller and more concise than their Python and non-UDF SQL code.

Besides the in-house experience where YeSQL is the core tool of trade for the OpenAIRE data scientists, external practitioners and data scientists have also been using YeSQL (including its first incarnation named madIS) to facilitate their research. To the best of our knowledge, to date the YeSQL language has been used by professionals in several domains such as geospatial ontologies [3, 4, 23, 38], text mining and information extraction [20, 42], data cleaning and exploration [24], and medical machine learning [74].

7 CONCLUSIONS

In this paper, we have presented YeSQL, an SQL extension with rich UDF support along with a pluggable architecture to easily integrate it with either server-based or embedded database engines. YeSQL supports Python UDFs fully integrated with relational queries as scalar, aggregator, or table functions. The YeSQL language has been designed to increase the productivity of data scientists and analysts in developing complex algorithms within an extended relational model. YeSQL outperforms alternative implementations due to its several performance enhancements, including tracing JIT compilation of Python UDFs, parallelism and fusion of UDFs, stateful UDFs, and seamless integration with a database engine.

YeSQL has been deployed in production and is being used by data scientists in several domains. We actively pursue various future directions including extensions to federated, heterogeneous systems and optimization opportunities in UDF fusion and query rewriting.

ACKNOWLEDGMENTS

This work is supported by EU's Horizon 2020 projects OpenAIRE-Advance, INODE, and HBP with grant agreement numbers 777541, 863410, and 945539, respectively.

REFERENCES

[1] Apache Arrow. 2022. Available at: https://arrow.apache.org/.

[2] Stefan Behnel, Robert Bradshaw, Craig Citro, Lisandro Dalcín, Dag Sverre Seljebotn, and Kurt Smith. 2011. Cython: The Best of Both Worlds. *Comput. Sci. Eng.* 13, 2 (2011), 31–39.

[3] Konstantina Bereta, Hervé Caumont, Ulrike Daniels, Erwin Goor, Manolis Koubarakis, Despina-Athanasia Pantazi, George Stamoulis, Sam Ubels, Valentijn Venus, and Firman Wahyudi. 2019. The Copernicus App Lab project: Easy Access to Copernicus Data. In *EDBT*. 501–511.

[4] Konstantina Bereta, Hervé Caumont, Erwin Goor, Manolis Koubarakis, Despina-Athanasia Pantazi, George Stamoulis, Sam Ubels, Valentijn Venus, and Firman Wahyudi. 2018. From Copernicus Big Data to Big Information and Big Knowledge: A Demo from the Copernicus App Lab Project. In *CIKM*. 1911–1914.

[5] Mark Blacher, Joachim Giesen, SÃűren Laue, Julien Klaus, and Viktor Leis. 2022. Machine Learning, Linear Algebra, and More: Is SQL All You Need?. In *CIDR*.

[6] Carl Friedrich Bolz, Antonio Cuni, Maciej Fijalkowski, and Armin Rigo. 2009. Tracing the meta-level: PyPy's tracing JIT compiler. In *ICOOOLPS*.

[7] CFFI. 2022. Using CFFI for embedding. Available at: https://cffi.readthedocs.io/en/latest/embedding.html.

[8] CFFI. 2022. Using the ffi/lib objects. Available at: https://cffi.readthedocs.io/en/latest/using.html#extern-python-new-style-callbacks.

[9] CFFI Docs. 2022. Using the ffi/lib objects. Available at: https://cffi.readthedocs.io/en/latest/using.html.

[10] Hanfeng Chen, Joseph Vinish D'silva, Hongji Chen, Bettina Kemme, and Laurie Hendren. 2018. HorseIR: Bringing Array Programming Languages Together with Database Query Processing. In *ACM SIGPLAN DLS*. 37âĂŞ49.

[11] Hanfeng Chen, Joseph Vinish D'silva, Laurie J. Hendren, and Bettina Kemme. 2021. HorsePower: Accelerating Database Queries for Advanced Data Analytics. In *EDBT*. 361–366.

[12] Alvin Cheung, Armando Solar-Lezama, and Samuel Madden. 2013. Optimizing database-backed applications with query synthesis. In *SIGPLAN*. 3–14.

[13] Databases and Information Systems Group at TU Ilmenau. 2022. Compiled UDF engine. Available at: https://github.com/dbis-ilm/Compiled_UDF_engine/tree/main/exp.

[14] Joseph Vinish D'silva, Florestan De Moor, and Bettina Kemme. 2018. AIDA - Abstraction for Advanced In-Database Analytics. *PVLDB* 11, 11 (2018), 1400–1413.

[15] DuckSB. 2022. Available at: https://duckdb.org/.

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

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

[18] Yannis Foufoulas, Alkis Simitsis, Lefteris Stamatogiannakis, and Yannis Ioannidis. 2022. The YeSQL language. Available at: https://github.com/athenarc/YeSQL/blob/main/Specs.md.

[19] Yannis Foufoulas, Lefteris Stamatogiannakis, Harry Dimitropoulos, and Yannis Ioannidis. 2017. High-pass text filtering for citation matching. In *TPDL*.

[20] Yannis Foufoulas, Lefteris Stamatogiannakis, Harry Dimitropoulos, and Yannis Ioannidis. 2017. High-pass text filtering for citation matching. In *International Conference on Theory and Practice of Digital Libraries*. Springer, 355–366.

[21] Tasos Giannakopoulos, Yannis Foufoulas, Harry Dimitropoulos, and Natalia Manola. 2019. Interactive Text Analysis and Information Extraction. In *IRCDL*.

[22] Theodoros Giannakopoulos, Eleftherios Stamatogiannakis, Ioannis Foufoulas, Harry Dimitropoulos, Natalia Manola, and Yannis Ioannidis. 2013. Content visualization of scientific corpora using an extensible relational database implementation. In *TPDL*.

[23] Konstantinos Giannousis, Konstantina Bereta, Nikolaos Karalis, and Manolis Koubarakis. 2018. Distributed Execution of Spatial SQL Queries. In *IEEE Big Data*. 528–533.

[24] Anna Gogolou, Marialena Kyriakidi, and Yannis E. Ioannidis. 2016. Data exploration: a roll call of all user-data interaction functionality. In *ExploreDB*. 31–33.

[25] Google groops. 2022. Not releasing the GIL. Available at: https://groups.google.com/g/python-cffi/c/ytAHJ0_YYAs/m/lUdP8TWPDQAJ.

[26] GraalVM. 2022. Available at: https://www.graalvm.org.

[27] Lars Grammel. 2020. Wrangling US Flight Data - Part 1. 2015. Available at: https://www.trifacta.com/blog/wrangling-us-flight-data-part-1.

[28] Surabhi Gupta, Sanket Purandare, and Karthik Ramachandra. 2020. Aggify: Lifting the Curse of Cursor Loops using Custom Aggregates. In *SIGMOD*. 559–573.

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

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

[31] Anna Herlihy, Periklis Chrysogelos, and Anastasia Ailamaki. 2022. Boosting Efficiency of External Pipelines by Blurring Application Boundaries. In *CIDR*.

[32] Alekh Jindal, K. Venkatesh Emani, Maureen Daum, Olga Poppe, Brandon Haynes, Anna Pavlenko, Ayushi Gupta, Karthik Ramachandra, Carlo Curino, Andreas Mueller, Wentao Wu, and Hiren Patel. 2021. Magpie: Python at Speed and Scale using Cloud Backends. In *CIDR*.

[33] Johannes Kepler University. 2022. The Truffle Language Implementation Framework. Available at: http://www.ssw.uni-linz.ac.at/Research/Projects/JVM/Truffle.html.

[34] Alfons Kemper and Thomas Neumann. 2011. HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots. In *ICDE*.

[35] 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.

[36] 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.

[37] Steffen KlÃďbe, Robert DeSantis, Stefan Hagedorn, and Kai-Uwe Sattler. 2022. Accelerating Python UDFs in Vectorized Query Execution. In *CIDR*.

[38] Kostis Kyzirakos, Dimitrianos Savva, Ioannis Vlachopoulos, Alexandros Vasileiou, Nikolaos Karalis, Manolis Koubarakis, and Stefan Manegold. 2018. GeoTriples: Transforming geospatial data into RDF graphs using R2RML and RML mappings. *J. Web Semant.* 52-53 (2018), 16–32.

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

[40] Chris Lattner, Jacques A. Pienaar, Mehdi Amini, Uday Bondhugula, River Riddle, Albert Cohen, Tatiana Shpeisman, Andy Davis, Nicolas Vasilache, and Oleksandr Zinenko. 2020. MLIR: A Compiler Infrastructure for the End of Moore's Law. *CoRR* abs/2002.11054 (2020).

[41] Lua JIT, FFI library. 2022. Available at: https://luajit.org/ext_ffi.html.

[42] Natalia Manola. 2019. Interactive Text Analysis and Information Extraction. In *Digital Libraries: Supporting Open Science: 15th Italian Research Conference on Digital Libraries, IRCDL 2019, Pisa, Italy, January 31–February 1, 2019, Proceedings*, Vol. 988. Springer, 340.

[43] Kevin Modzelewski. 2014. Introducing Pyston: an upcoming, JIT-based Python implementation. Available at:. https://nuitka.net

[44] Kevin Modzelewski. 2020. Pyston v2: 20% faster Python. Available at:. https://blog.pyston.org/2020/10/28/pyston-v2-20-faster-python

[45] MonetDB. 2022. Base Types. Available at: https://www.monetdb.org/documentation-Jan2022/user-guide/sql-manual/data-types/base-types.

[46] MonetDB. 2022. Index Definitions. Available at: https://www.monetdb.org/documentation-Jan2022/user-guide/sql-manual/data-definition/index-definitions.

[47] MonetDB docs. 2022. Jit C UDFs. Available at: https://www.monetdb.org/documentation-Jan2022/user-guide/blog-archive/jit-c-udfs.

[48] Nuitka the Python Compiler. 2022. Available at: https://nuitka.net.

[49] Numpy. 2022. Available at: https://numpy.org/.

[50] Shoumik Palkar, James J. Thomas, Anil Shanbhag, Malte Schwarzkopf, Saman P. Amarasinghe, and Matei Zaharia. 2017. A Common Runtime for High Performance Data Analysis. In *CIDR*.

[51] Arash Partow. 2020. The Global Airport Database. Available at: https://www.partow.net/miscellaneous/airportdatabase.

[52] Richard Plangger and Andreas Krall. 2016. Vectorization in PyPy's Tracing Just-In-Time Compiler. In *SCOPES*. 67âĂŞ76.

[53] PostgreSQL. 2022. Procedural languages. In *PostgreSQL User Manual*.

[54] PostgreSQL Documentation. 2022. Create type. Available at: https://www.postgresql.org/docs/14/sql-createtype.html.

[55] Martin Prammer, Suryadev Sahadevan Rajesh, Junda Chen, and Jignesh Patel. 2022. Introducing a Query Acceleration Path for Analytics in SQLite3. In *CIDR*.

[56] PyPY. 2022. PyPy's Python packages compatibility. Available at: http://packages.pypy.org.

[57] PyPY. 2022. Python compatibility. Available at: https://www.pypy.org/compat.html.

[58] PyPY. 2022. Unicode strings. Available at: https://www.pypy.org/posts/2019/03/pypy-v71-released-now-uses-utf-8-451324088028792912.html.

[59] Mark Raasveldt and Hannes Mühleisen. 2016. Vectorized udfs in column-stores. In *SSDBM*.

[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] Armin Rigo and Maciej Fijalkowski. 2021. CFFI documentation.

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

[63] 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.

[64] Scipy. 2022. Available at: https://scipy.org.

[65] Weisong Shi, Jie Cao, Quan Zhang, Youhuizi Li, and Lanyu Xu. 2016. Edge Computing: Vision and Challenges. *IEEE Internet Things J.* 3, 5 (2016), 637–646.

[66] Varun Simhadri, Karthik Ramachandra, Arun Chaitanya, Ravindra Guravannavar, and S. Sudarshan. 2014. Decorrelation of user defined function invocations in queries. In *ICDE*. 532–543.
[67] Spark documentation. 2022. User Defined Aggregate Functions. Available at: https://spark.apache.org/docs/latest/sql-ref-functions-udf-aggregate.html.
[68] Leonhard Spiegelberg, Rahul Yesantharao, Malte Schwarzkopf, and Tim Kraska. 2021. Tuplex: Data Science in Python at Native Code Speed. In *SIGMOD*.
[69] SQLite C Interface. 2022. Create Or Redefine SQL Functions. Available at: https://www.sqlite.org/c3ref/create_function.html.
[70] SQLite documentation. 2022. Application-Defined SQL Functions. Available at: https://www.sqlite.org/appfunc.html.
[71] TPC-H Benchmark. 2022. Available at: http://www.tpc.org/tpch.
[72] Itamar Turner-Trauring. 2022. Faster string processing in Pandas. Available at: https://www.gresearch.co.uk/article/faster-string-processing-in-pandas.
[73] U.S. Department of Transportation Bureau of Transportation Statistics. 2020. Carrier history lookup table. Available at: https://www.transtats.bts.gov/Download_Lookup.asp?Lookup= L_CARRIER_HISTORY.
[74] Evert H Pieter Van Dijkhuizen, Orfeas Aidonopoulos, Nienke M Ter Haar, Denise Pires Marafon, Silvia Magni-Manzoni, Yannis E Ioannidis, Lorenza Putignani, Sebastiaan J Vastert, Clara Malattia, Fabrizio De Benedetti, et al. 2018. Prediction of inactive disease in juvenile idiopathic arthritis: a multicentre observational cohort study. *Rheumatology* 57, 10 (2018), 1752–1760.
[75] Vertica documentation. 2022. UDAF Class Overview. Available at: https://www.vertica.com/docs/10.0.x/HTML/Content/Authoring/ ExtendingVertica/UDx/AggregateFunctions/UDAFClassOverview.htm.