YeSQL: Rich User-Defined Functions without the Overhead

Yannis Foufoulas
University of Athens
Athens, Greece
johnfouf@di.uoa.gr

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

Yannis Ioannidis
University of Athens
Athens, Greece
yannis@di.uoa.gr

ABSTRACT

The diversity and complexity of modern data management applications 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 even more central in many applications spanning data science, data analytics, etc. Still, a critical limitation of UDFs, which to some extent has turned data scientists towards NoSQL systems, is the impedance mismatch between their evaluation and relational processing. 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. We currently support UDFs written in Python, which are fully integrated with relational queries as scalar functions, aggregators, or table returning functions. Key novel characteristics of YeSQL include easy implementation of complex algorithms, tracing JIT compilation of Python UDFs, and seamless integration with a database engine. Our demonstration will showcase (a) the usability and expressiveness of our approach, and (b) that our techniques of minimizing context switching between the relational engine and the Python VM are very effective and achieve significant speedups in common, practical use cases.

PVLDB Reference Format:
Yannis Foufoulas, Alkis Simitsis, and Yannis Ioannidis. YeSQL: Rich User-Defined Functions without the Overhead . PVLDB, 15(12): 3730 - 3733, 2022.
doi:10.14778/3554821.3554886

1 INTRODUCTION

Modern trends in data processing are characterized by a diversity of data sources and complex processing tasks executed on large volumes of data. This falls naturally within the scope of relational databases, which are extremely powerful data processing and data storage 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 UDFs are supported by most data processing systems, but currently have several limitations on their *usability* and *performance*. For example, MonetDB natively supports vectorized Python UDFs using Numpy, but these require static definition of their returned schema, as only table UDFs may return multiple rows. In addition, known performance enhancements such as a Just-In-Time (JIT) compiler are not used. The functions run on PythonâĂŹs 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 also supports Python UDFs without a 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. On the other hand, research prototypes of data engines that support Python UDFs, such as Tuplex [5], consider performance improvements (e.g., JIT compiler), but lack usability features (e.g., stateful, parametric functions) that would render these approaches more practical in real-world applications.

In this paper, we present *YeSQL*, an SQL extension and its implementation that provides more *usable* and more *performant* Python UDFs, and can be integrated into both server-based and embedded DBMSs. YeSQL 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.

Usability. YeSQL extends SQL with an alternative, equivalent syntax that affords compact expressions of many 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 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 CPythonâĂŹs 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. YeSQL uses PyPy [1] as its tracing JIT compiler and CFFI [4] 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. Note that the YeSQL performance enhancements are *orthogonal* to the YeSQL syntax mentioned earlier; that is, YeSQL boosts Python UDFs either in YeSQL queries and/or in regular SQL queries.

Proceedings of the VLDB Endowment, Vol. 15, No. 12 ISSN 2150-8097.
doi:10.14778/3554821.3554886

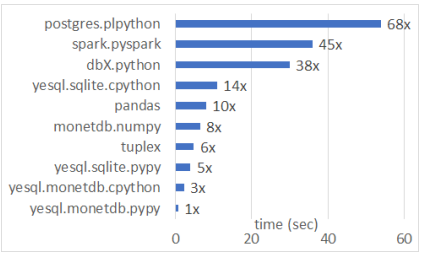


Figure 1: Performance of a scalar UDF on a string column

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 [5], Spark/PySpark, Pandas, and YeSQL extensions to MonetDB and SQLite with CPython and PyPy. In this comparison, we used real estate data from Zillow and implemented a Python scalar UDF which gets a column with the description of an apartment (example value: '2 bds, 1.0 ba, 856 sqft') and 7 million rows. The UDF extracts and returns an integer representing the number of bedrooms. Figure 1 shows that JIT-compiled YeSQL implementations allow MonetDB and SQLite to run 6x to 68x faster than the other candidates.

Deployment. YeSQL is designed to enable data scientists develop and run general-purpose algorithms in SQL seamlessly. It is currently used in production by OpenAIRE (openaire.eu), 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, text mine Open Access publications, and extract links to funders, software, citations, 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. To the best of our knowledge, YeSQL has also been used by data scientists in other domains as well, such as geospatial ontologies, text mining and information extraction, data cleaning and exploration, and medical machine learning [2].

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. A full description of the YeSQL design and implementation, and extensive experiments, can be found in [2]. In this paper, we focus on the key points of YeSQL and the proposed demonstration scenarios.

2 DEMONSTRABLE FEATURES

2.1 Architecture and Integration

Architecture. YeSQL can be integrated with either a server-based DBMS (e.g., MonetDB) or an embedded DBMS (via SQLITE API). Figure 2a shows the core components of YeSQL architecture.

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 2a) 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. SQL queries using standard SQL syntax simply pass through. When integrated with a server-based DBMS, CFM first compiles the UDFs so that they are accessible by the DBMSâĂŹs 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.

Integration. YeSQL works as a modular addition to a DBMS and it is compatible with all popular operating systems. For a server-based DBMS, we leverage 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 for each produced value.

Implementation. The YeSQL codebase is 66K lines of Python and C++, including 18.5K lines for the code definitions of 150+ Python UDFs currently supported.

2.2 Functionality Overview

YeSQL handles 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 Python UDFs that use pre-existing Python libraries (e.g., numpy, nltk) via `import`, thus inheriting features that are commonly used by data scientists.

Demo Scenario 1. The keystone principle of YeSQL is to enable data scientist develop and run algorithms in SQL seamlessly. Our demo script includes the step-by-step implementation of an example algorithm: Given a table with all NSF project grant identifiers (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 version of this algorithm can be expressed 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 a 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 random lines from the file (`sample 100`). Its returned schema depends on the stored data. In this case, it returns 2 columns named 'id' and 'text'. It processes each text with scalar function keywords (`keywords(text)`) which removes punctuation marks using a precompiled pattern. It runs a scrolling window over the text (`textwindow(keywords(text),10, 1, 5, '\d{7}')`). This function returns 3 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. Using standard join, 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 filtering is 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% accuracy in OpenAIRE, a real-world application.

The purpose of this demo scenario is to illustrate the key 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 *row 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 composition of UDFs in a functional language style. The query fragment: "`sample 100 file 'publications.json'`" first reads the file containing the publications and then gets a random sample with 100 rows.

2.3 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, and (b) data conversion overhead, data is represented differently in the two environments and need to be wrapped/unwrapped 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.

Tracing JIT. 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. YeSQL employs the PyPy dynamic JIT compiler. 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 several optimizations in a traditional static compiler. By fusing UDFs and exposing larger chunks of Python code, YeSQL allows PyPy to perform better compilation.

PyPy also facilitates the integration with a DBMS. For example, in MonetDB that supports vectorized UDFs, the pointer to the whole column is passed with one function call, minimizing multiple function calls overhead via CFFI. And since PyPy enables its own vectorization [3] these conversions are transparent to the user. 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:

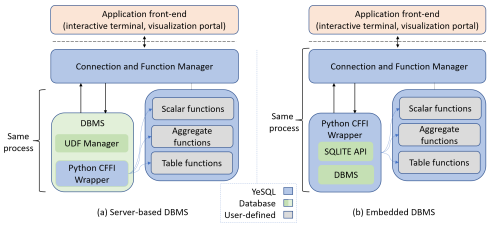
```
def lenstr_wrapped(input,insize,result):       |  def lenstr(val):
  for i in range(insize):                      |    return len(val)
    result[i] = lenstr(ffi.string(input[i]))   |
  return 1                                     |
```

Function *lenstr_wrapped* is embedded and called by the DBMS. 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*. *lenstr* is the UDF that the data scientist actually implements.

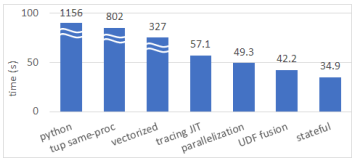
CFFI wrapper works a little differently with an embedded DB. It submits the UDF as a function callback with the appropriate conversions and the function is called by the database.

Seamless integration with the DBMS. In YeSQL, UDFs are wrapped using embedded CFFI and PyPy. At UDF execution, the data is transferred to CFFI as pointers to *cdata objects* without any data copies. Integers and float columns are used directly by PyPy. For string columns, we have three options: (a) *ffi.string* transforms the string to a format that is understandable by PyPy, (b) *ffi.buffer* returns a Python *memoryview* (i.e., an array of characters) without copying the string; and (c) *direct pass*, which passes directly the pointer to the C string enabling low level optimizations.

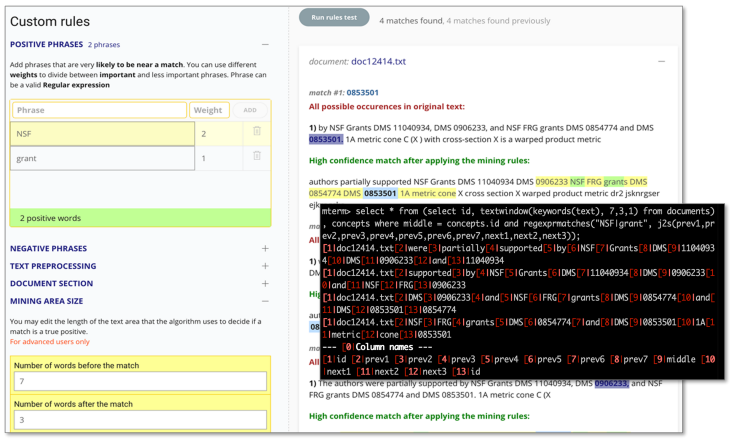
UDF Fusion. When more than one UDF run in sequence and can be fused, the fusion takes place at the level of CFFI wrapper function.



(a) YeSQL: Architecture options



(b) Main factors in boosting Python UDF execution



(c) Interactive interface for dynamic tuning of mining algorithms

Figure 2: YeSQL Architecture, performance enhancements, and example application front-end

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 if not fused they run on a different trace. By fusing them, we expose longer sequences of instructions so more optimization is enabled by the tracing JIT. YeSQL supports fusion of scalar, aggregate and table functions [2].

Parallelism. The performance of Python UDFs running 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. 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). However, in PyPy, the creation of a Python object is faster as less memory is required to create and store a PyObject. 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.

Stateful UDFs. Additionally, stateful UDFs may enable some specific optimizations (e.g. precompile 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 any state is defined outside them). When running on a vectorized UDF like in MonetDB, the state is also available during the processing of different rows since the whole column is passed to the CFFI wrapper.

Demo Scenario 2. We will demonstrate the effectiveness of each of these techniques using a query with four UDFs over the Zillow dataset. The query will first run as a spawned process as tuple-at-a-time (this resembles an out-of-the-box execution on PostgreSQL with PL/Python). Next, we will try the boosting techniques one at a time, as follows: (a) vectorized execution using embedded NumPy, (b) tracing JIT-compilation, (c) parallelism, (d) UDF fusion on JIT, and (e) stateful UDF execution. Figure 2b presents a performance breakdown illustrating the extent that each technique contributes to performance. We will also show that although any of these techniques in isolation helps boosting UDF execution, applying them all and in a specific order increases the optimization opportunities.

3 OUR PRESENTATION

Our presentation script is designed for expert and novice users, with or without experience in expressing complex algorithms as UDFs. We will showcase YeSQL with representative, data science pipelines over three datasets: zillow, flights, and text-mining.

Use Cases. Our presentation starts with the two demo scenarios presented earlier to showcase YeSQL usability and performance. For usability, we will also present step-by-step algorithm implementation with YeSQL, inspired by our experience with OpenAIRE. For performance, we will also examine: Python UDFs in embedded SQLite and MonetDB with embedded Python UDFs. In both setups, the UDFs will be executed with CPython's interpreter and PyPy's JIT compiler. Canned examples as the one in Figure 1 will be available for testing on YeSQL, Tuplex, MonetDB, PostgreSQL, SQLLite, and dbX, covering aspects such as varying data sizes and parallelization, UDF fusion, parallelism, scalability, and resource usage.

User interaction. For off-script presentation, the audience will try existing UDFs or create and execute their own Python UDFs on real data. We will enable interaction with YeSQL either directly via a terminal (which implements useful features like autocomplete, help, history and more) or via a GUI that enables YeSQL query execution and UDF submission. YeSQL with its dynamic nature allows the development of interactive interfaces for the implementation and dynamic tuning of algorithms using polymorphic functions. Figure 2c shows an example of such an interface that provides a possible implementation of 'NSF' mining. The back-end translates the user's input directly into a YeSQL query (shown in the terminal snippet) and executes it using the predefined polymorphic UDFs.

ACKNOWLEDGMENTS

This work is supported by EU's Horizon 2020 projects with grant agreement numbers 101017452, 863410, 945539, and 955895.

REFERENCES

[1] Bolz, C. F., Cuni, A., Fijalkowski, M., and Rigo, A. Tracing the meta-level: PyPy's tracing JIT compiler. In *ICOOOLPS* (2009).
[2] Foufoulas, Y., Simitsis, A., Stamatogiannakis, L., and Ioannidis, Y. YeSQL: "You extend SQL" with rich and highly performant user-defined functions in relational databases. In *VLDB* (2022).
[3] Plangger, R., and Krall, A. Vectorization in PyPy's Tracing Just-In-Time Compiler. In *SCOPES* (2016).
[4] Rigo, A., and Fijalkowski, M. CFFI documentation, 2021.
[5] Spiegelberg, L., Yesantharao, R., Schwarzkopf, M., and Kraska, T. Tuplex: Data science in python at native code speed. In *SIGMOD* (2021).