YeSQL with Rich and Highly Performant UDFs in RDBs

DT-DB42-M: The Question to or the Better Answer on 42? (Summer term 2023)

University of Bamberg
Bamberg, Germany
jinghao.wu@stud.uni-bamberg.de

Jinghao Wu

ABSTRACT

With the development of modern data management applications, the data source has become more and more varied and complex. It leads to the popularity of user-defined Functions(UDFs), also the extension of relational paradigms which can syntactically and semantically support them, even in a traditional database management system(DBMS). But there is a typical limitation of UDFs which is the impedance mismatch between evaluation and relational processing. In this seminar paper, we present YeSQL, an SQL extension, which can optimize execution performance in this context. It can be easily integrated with either server-based or embedded DBMS. The outstanding characteristics of YeSQL include easy implementation of complex algorithms and five performance enhancements. Then, the evaluation analysis demonstrates the efficiency of query execution in alternative combinations of environment, database engines, and techniques.

1 INTRODUCTION

As data sources and forms have become more complex, forms of database management systems have been developed to accommodate various needs. In this case, we discuss the performance of YeSQL based on the two representative DBMSs.

1.1 MonetDB

MonetDB is a column-oriented open source relational DBMS. It is designed to provide high performance for complex queries in large databases, such as tables with hundreds of columns and millions of rows. MonetDB has been used for high-performance applications such as online analytical processing, data mining, geographic information systems (GIS), resource description frameworks (RDF). MonetDB now supports UDFs written in Python/NumPy. The implementation uses Numpy arrays and therefore has limited overhead - and provides functional Python integration with native SQL functions to match the speed.

1.2 SQLite

SQLite is a database engine written in the C programming language. It is a standalone application, but rather a library that software developers embed into their applications. SQLite is designed to allow it to run without the need to install a database management system or require a database administrator. Unlike a client-server database management system, the SQLite engine does not have a separate process with which the application communicates. Instead, the linker integrates the SQLite library - statically or dynamically - into an application that uses SQLite functionality through simple

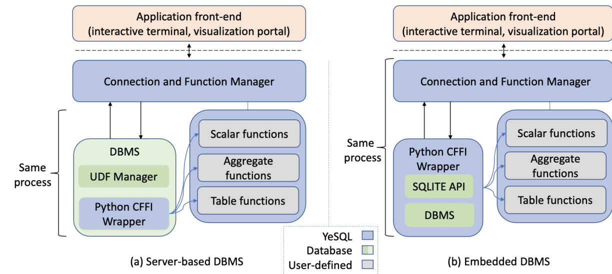

Figure 1: System Architecture of YeSQL[1]

function calls, reducing the latency of database operations; for simple queries with low concurrency, SQLite performance benefits from avoiding the overhead of inter-process communication.

In this seminar paper, we describe in detail the system architecture of YeSQL and how it functions and is implemented. About YeSQL, this SQL extension provides more expressive and powerful Python UDFs. It enriches SQL with three types of Python UDFs, scalar, aggregate, and table functions.

2 YESQL

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).[1]

2.1 System Architecture

We define two groups of user roles. First, the application users like data analysts submit their queries to the interactive terminal or visualization interface, which delivers queries to the Connection and Function Manager further. UDF developers create their UDFs (gray boxes in Figure 1) and YeSQL loads them into the DBMS. Possibly, one user fulfills both roles

The Connection and Function Manager (CFM) consists of three components: a)Parser: Check whether the query YeSQL or standard SQL. When queries use standard syntax, then simply pass through. b)Code generator: It transforms the query to a specific language depending on the underlying DBMS. c)Function manager: Submit UDFs to the DBMS. It accesses the packages which define the UDF.

Submission of UDFs depends on whether YeSQL is integrated with an embedded or a server-based DBMS. When integrated with a server-based DBMS(Left side in Figure 1), CFM first complies the UDFs, then they can be accessible by the DBMS's UDF manager as an in-process embedded library. It submits declarations directly

to the DBMS. When integrated with an embedded DBMS(Right side in Figure 1), it submits the UDFs via the Python CFFI wrapper. The UDFs are executed in the same process with the CFM and the DBMS. These three gray boxes are typical types of UDFs which are classified into scalar, aggregate, and table functions. It provides flexibility and expressive power in data processing and analysis. The Python CFFI wrapper from both sides is the layer that are borderlines between Python and the database engine. With server-based DBMS, it directly calls Python UDFs from the shared library. With an embedded DBMS, it submits the UDFs as callback functions and makes sure that the seamless data exchange between DBMS and Python UDF is guaranteed. SQLite API natively supports extended-SQL functionality through C UDFs.

2.2 Characteristics of YeSQL

2.2.1 Usability and Expressiveness. For this characteristic, the author introduces functionality, syntactic inversion, and code generation of YeSQL by giving some corresponding examples. YeSQL offers support for table function chaining which means the queries can be nested for each table returning UDF. The example from the paper lists three UDFs from the PostgreSQL query. (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'''')'')');
```

After using the syntax inversion, the query is much easier as we can see as follow:

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

Another example shows a small test in practice. 380 undergraduate students were required to complete an assignment with and without SQL. The task was about the development of two algorithms and in the end approximately 86% of students finished successfully. After the experiment, the feedback was quite well. Most students said programming with YeSQL was easy and they liked that YeSQL code was more concise than Python and non-UDF SQL code.

2.2.2 Performance. Performance is the reason we keep pursuing faster algorithms, and better technology. YeSQL aims to avoid the impedance mismatch between the relational SQL evaluation and the procedural Python execution. This mismatch causes two major overheads when implementing YeSQL. (a)Context switching: frequent execution can be expensive if one facility needs to call another facility through various levels of indirection. (b) Data conversion: Because of different data forms in two execution environments, we need to be encoded/decoded or wrapped/unwrapped. To reduce or remove these expenses, YeSQL offers five techniques to improve performance, which are tracing JIT compilation, seamless integration with the DBMS, UDF fusion, parallelism, and support for stateful UDFs. Figure 2 presents a performance classification illustrating the extent that each technique contributes to performance. While any one of these techniques can help improve UDF execution, applying them all in a specific order increases the chances of optimization.

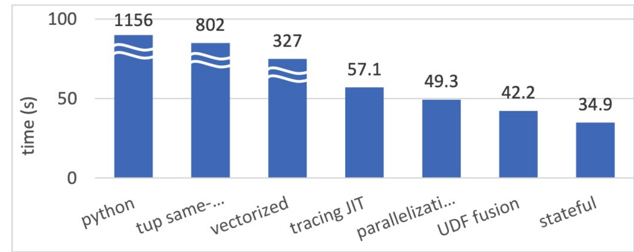

Figure 2: Largest factors in boosting Python UDF execution[1]

a) JIT(Just-in-Time)-compiled UDFs: JIT compilation enhances program performance by dynamically compiling portions of a program into machine code during runtime. It highlights the difference between method-based JIT compilers and tracing JIT compilers, which focus on frequently executed loops. The authors propose the use of the PyPy dynamic compiler, a high-performing engine for Python program execution, in conjunction with the YeSQL query compilation as a pre-optimization step. The integration of PyPy with a DBMS is discussed, specifically the compatibility of CFFI's internal array representation with Numpy, which aligns well with the Python support of MonetDB. This integration allows for seamless passing of arrays between PyPy and CPython UDFs, minimizing overheads. In summary, PyPy with YeSQL can optimize UDFs in Python programs.

b) Seamless Integration with DBMSs: UDFs are wrapped using embedded CFFI. During UDF execution, data is transferred to CFFI as pointers to cdata objects without any data copies. For integer and float columns, Python uses them directly. However, for string columns, three options are presented: using ‘ffi.string‘ to transform strings into a format understandable by PyPy or CPython, using ‘ffi.buffer‘ to return a memory view of the string without copying it, or passing the pointer to the C string directly, allowing for low-level optimizations by manipulating the pointer in a C-like manner.

c) UDF fusion: The practice of producing small and reusable UDFs is common and beneficial for productivity. These UDFs can be combined in workflows, such as text mining, where tasks like tokenization, stemming, and normalization are performed sequentially. To enhance performance, UDF fusion can be employed. This involves combining multiple UDFs into a single function at the CFFI wrapper level. The fused UDFs eliminate CFFI conversions and allow for longer instruction sequences, enabling more optimization by tracing JIT.

d) UDF parallelization: Parallelism is generally beneficial for improving program performance and scalability. However, in the case of Python UDFs within DBMSs, their parallel execution is limited by the Global Interpreter Lock (GIL). The GIL is a lock that allows only one thread to execute Python code at a time, preventing deadlocks. While the GIL itself doesn't add significant overhead, it becomes a performance bottleneck in CPU-bound and multi-threaded scenarios. The GIL is also active during the creation of Python objects when translating database data for use in Python UDFs. Releasing and acquiring the GIL incur additional costs. In the case of CFFI, the GIL is released lazily without synchronization,

which allows for optimized performance. However, in PyPy, object creation is faster and requires less memory, leading to improved GIL release and acquisition.

e) stateful UDFs: Data processing systems primarily support stateless UDFs, where only the output persists beyond execution. However, stateful UDFs offer advantages in algorithm development and performance optimization. Embedded DBMSs allow stateful UDFs with access to external states and server-based DBMSs like MonetDB share states across different UDFs. Developers can provide UDFs as Python modules, enabling them to import packages and perform costly operations at the global scope. This improves performance and allows for pre-compiling patterns and performing other optimizations. While stateless UDFs are more common, stateful UDFs provide interesting opportunities for data analysis and data science tasks.

2.3 Portability and Modularity

2.3.1 Portability. YeSQL is an architecture that enhances existing systems, particularly those using the SQLite API, which accelerates queries in SQLite using Apache Arrow for analytics. Also, it extends the functionality to work in server-based architectures and is compatible with works like Hustle[3].

2.3.2 Modularity. YeSQL is a modular addition to a DBMS that can be easily installed and is compatible with various operating systems. It integrates with data processing systems through C UDFs and ODBC, adapting the Python CFFI wrapper component to meet specific requirements. YeSQL UDFs run within the same process as the DBMS's C-UDF API, either in-process or out-of-process. For server-based DBMSs, YeSQL leverages the DBMS's execution model. For example, in MonetDB which has a vectorized execution model, the data is passed via CFFI with one function call as array pointers. With embedded databases like SQLite, YeSQL utilizes the streaming architecture and Python generators to handle large datasets efficiently, allowing for the creation of data pipelines.

2.4 Deployment

YeSQL is used by OpenAIRE3[1], a technical infrastructure involving a consortium of 65 European universities, research centers, and other institutions. It is employed for various tasks, including harvesting research data from multiple sources, text mining open-access publications, and extracting relevant links. OpenAIRE3 has successfully harvested millions of publications, datasets, research software artifacts, and projects from numerous funders. The infrastructure relies on over 150 YeSQL UDFs to accomplish these tasks efficiently.

3 EVALUATION

The YeSQL codebase consists of approximately 66K lines of Python and C++, including 18.5K lines dedicated to the definitions of over 150 Python UDFs currently supported. The evaluation of YeSQL involves three representative data science pipelines and micro-benchmarks of specific design features.

[1]OpenAIRE is an active network in 35 countries, the National Open Access Desks, who are experts in the local scene and are eager to help you on any issues related to open science.
https://www.openaire.eu

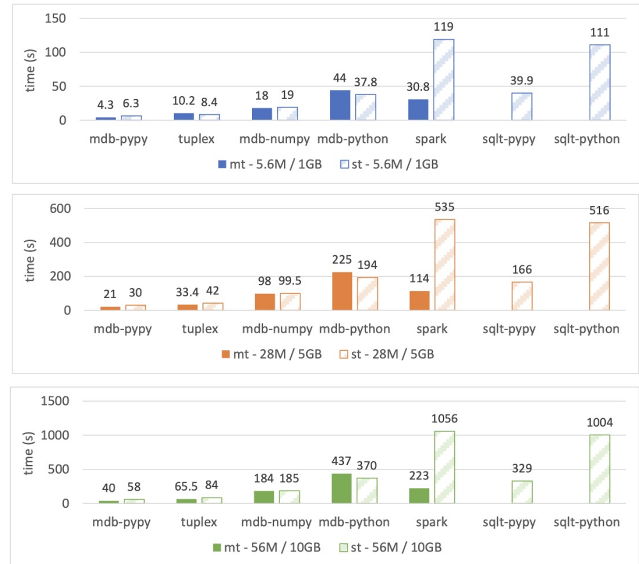

Figure 3: Zillow for varying data sizes and paralization[1]

3.1 Setup

3.1.1 Hardware and Software. The experimental setup includes running the experiments on an Intel Core i7-4930K processor with 64GB of RAM and Ubuntu 20.04 as the operating system. The measurements are performed with cold caches on SSD disks, and an average of five executions is reported. YeSQL is compared against Tuplex[4], MonetDB, PostgreSQL, a commercial distributed analytic database engine (dbX), Pandas, and Spark (PySpark).

3.1.2 Datasets. The datasets used for evaluation are the zillow, flights, and text-mining pipelines. [2, 4]The zillow and flights pipelines are obtained from Tuplex's GitHub repository, and the text-mining pipeline comes from a real-world application called OpenAIRE. The zillow dataset has three size variations: 1GB/5.6M rows, 5GB/28.6M rows, and 10GB/56M rows. The flights dataset has three size variations: 1.6GB/5M rows, 3.2GB/10M rows, and 6.4GB/20M rows.

3.2 End-to-end Pipeline Evaluation

In the end-to-end pipeline evaluation, YeSQL with tracing JIT on MonetDB (mdb.pypy) outperforms other candidates in both single-threaded and multi-threaded executions. MonetDB with CPython (mdb.python) suffers from slow parallelism due to the Global Interpreter Lock (GIL) and is slower than mdb.pypy. YeSQL is faster than Tuplex in both single-threaded and multi-threaded executions. SQLite performs well in single-threaded execution but lacks support for parallelism. Spark/PySpark has slower execution due to the spawning of a main Java process and separate Python processes. This superiority is observed in various datasets, different data sizes, and execution forms. (Shown in Figure 4,5,6)

3.3 Several Micro-experiments

In the micro-experiments, the tracing JIT in YeSQL significantly improves the performance of UDF execution compared to native

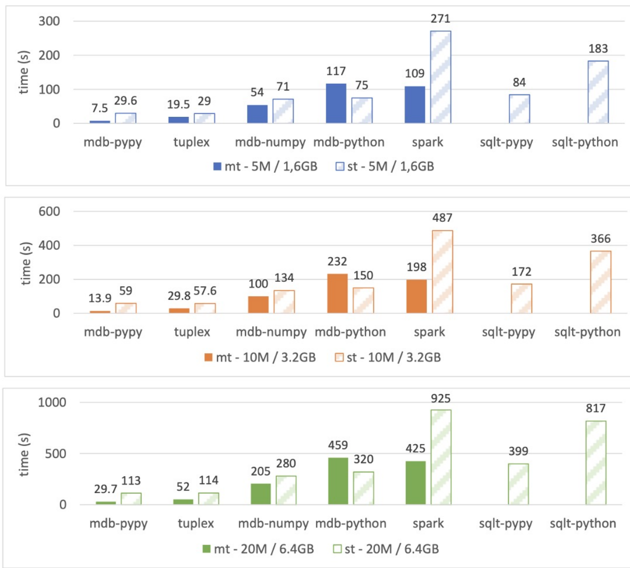

Figure 4: Flights pipeline[1]

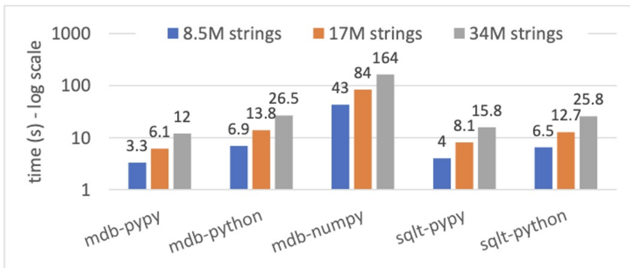

Figure 5: Text mining pipeline[1]

Python, Cython, Nuitka, and Numba implementations. YeSQL's seamless integration with MonetDB allows UDFs to achieve similar or better performance compared to native SQL queries. The overhead of transferring string columns using ffi.string compared to direct pass is evaluated, and the impact of different setups (caches, storage types, parallelization) on YeSQL's performance is examined.

Figure 6 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). Overall, YeSQL with tracing JIT on MonetDB demonstrates superior performance in both end-to-end pipelines and micro-experiments compared to other systems and frameworks.

4 CONCLUSION

YeSQL is a groundbreaking SQL extension outlined in this paper. It introduces an array of features to enhance UDF support, making it an ideal choice for data scientists and analysts. With YeSQL, developers can easily integrate it with both server-based and embedded database engines, thanks to its pluggable architecture. YeSQL supports Python UDFs fully integrated with relational queries as

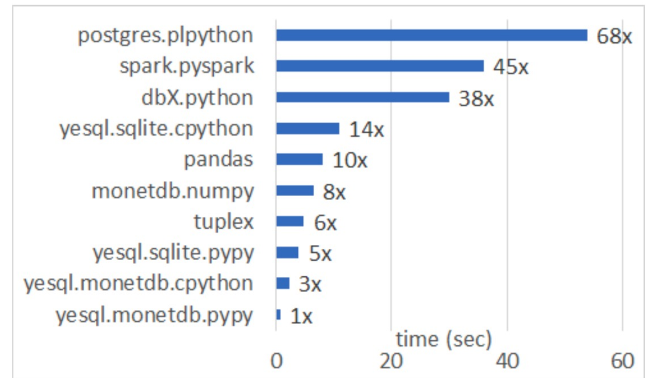

Figure 6: Performance of a scalar UDF on a string column[1]

scalar, aggregator, or table functions. YeSQL goes beyond alternative implementations by offering performance enhancements. These include tracing JIT compilation of Python UDFs, parallelism and fusion of UDFs, stateful UDFs, and seamless integration with database engines. When evaluating YeSQL with tracing JIT on MonetDB, it becomes evident that it outperforms other systems and frameworks in terms of overall performance. This superiority is observed in various scenarios, including end-to-end pipelines and micro-experiments. The practicality of YeSQL is validated through its deployment in production environments, where data scientists from diverse domains utilize its capabilities.

Furthermore, the research team is actively exploring future directions, such as extending YeSQL to federated and heterogeneous systems and optimizing UDF fusion and query rewriting.

REFERENCES

[1] Yannis Foufoulas, Alkis Simitsis, and Yannis Ioannidis. 2022. YeSQL. *Proceedings of the VLDB Endowment* 15, 12 (2022), 3730–3733. https://doi.org/10.14778/3554821.3554886

[2] Tasos Giannakopoulos, Yannis Foufoulas, and Harry Dimitropoulos. 2019. Interactive Text Analysis and Information Extraction. Zenodo. https://doi.org/10.5281/zenodo.4677595

[3] Chen Martin Prammer, Suryadev Sahadevan Rajesh and Patel. 2022. Introducing a Query Acceleration Path for Analytics in SQLite3. *Proceedings of the VLDB Endowment* 7 (2022), 1–7.

[4] Leonhard Spiegelberg, Rahul Yesantharao, Malte Schwarzkopf, and Tim Kraska. 2021. Tuplex: Data Science in Python at Native Code Speed. In *Proceedings of the 2021 International Conference on Management of Data (SIGMOD '21)*. Association for Computing Machinery, New York, NY, USA, 1718–1731. https://doi.org/10.1145/3448016.3457244