Accelerating Python UDFs in Vectorized Query Execution

Steffen Kläbe*
Actian
Ilmenau, Germany
steffen.klaebe@actian.com

Bobby DeSantis
Actian
Palo Alto, USA
robert.desantis@actian.com

Stefan Hagedorn
TU Ilmenau
Ilmenau, Germany
stefan.hagedorn@tu-ilmenau.de

Kai-Uwe Sattler
TU Ilmenau
Ilmenau, Germany
kus@tu-ilmenau.de

ABSTRACT

Modern analytical database systems offer support for user-defined funtions as a flexible extension to SQL. Python is one of the most popular UDF languages being easy to use and offering a rich feature set for data-intensive tasks, but also suffering from bad performance and scalability. In this work, we describe approaches to accelerate embedded Python UDF execution using vectorization, parallelisation and compilation. We compare different compilation frameworks and show how Python code can be compiled, dynamically loaded and queried during database runtime in a transparent way. Our evaluation showed that using compilation and parallelisation together leads to significant speedups for various use cases.

KEYWORDS

Python user-defined functions, Python compilation, Python parallelisation, Vectorized query execution

1 INTRODUCTION

User-defined functions (UDFs) play an important role in modern data analytics, as they offer a flexible, modular and reusable way to get business insights tailored for the user's needs. Many popular data warehouse systems integrate UDFs in different flavors to extend the functionality of the SQL frontend. Depending on the actual system, users can define UDFs using C/C++, Java, Python, Scala or Javascript code and use them in their SQL queries. Additionally, there are two types of UDFs, namely scalar UDFs, that produce a single result per input tuple, and table UDFs, that produce a new (potentially empty) table with an arbitrary number of results for a given input table. Python evolved to one of the most popular and important languages for data analytics over the recent years, which has several reasons. First, Python is easy to write and enables rapid prototyping, as a lot of challenges like typing are solved by the interpreter and are not a user's responsibility. Second, a large amount of frameworks were developed for data intensive tasks like data cleaning, data analytics and especially machine learning. This way, setting up and maintaining a database system, creating tables and loading data is not necessary to get fast insights in small datasets.

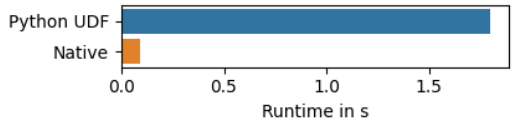

Figure 1: UDF vs. native performance for basic modulo operation on 8 million tuples

Consequently, Python has a wide range of users and makes it less important to be a SQL expert to analyse data.

Offering Python UDFs increases a database system's usability for a wide range of users. However, the major bottleneck of UDFs is scalability, as shown in the analysis of [5], where UDFs take the major part of the query runtime. This especially holds for Python UDFs due to several reasons. First, data needs to be converted from the internal database representation to a Python format and vice versa for the UDF results. Second, data needs to be transferred to the Python interpreter outside the database engine and results are sent back to the engine. Third, the interpreted code execution in the Python interpreter is typically slower than the execution of compiled and optimized code in the database engine. For operations that can be expressed both as a UDF and as a native database operator there is a large gap between the performance of a UDF and the respective native implementation of the operation, as shown in Figure 1.

Due to the broad functionality, Python UDFs are used by frontend frameworks like Grizzly [6] to offer support for advanced analytics. In this context, typical UDFs may be complex and might include external modules. In order to accelerate these kind of analytics, the main goal of this paper is to investigate the performance gap shown in Figure 1, aiming to reach native performance as much as possible. Figure 2 shows different levels of executing Python code from a database engine. While using the Python interpreter (1) is the most general approach, it also offers the worst performance and is therefore our upper baseline. On the other side, using plain C code (4) provides the best performance while restricting the support for Python modules and is therefore the lower baseline. Using the Python C-API (2) enables pre-compiling Python code in order to limit interpretation overhead while still providing full support of Python functionality. Using Numpy (3) is a special case here. While being fomulated as Python code, [13] shows that Numpy code can reach nearly native code speed as a consequence of a memory layout similar to C. However, UDFs are then limited to Numpy operations and fall back to (2) on the occurence of non-Numpy constructs.

*Work done while at TU Ilmenau.

 12th Annual Conference on Innovative Data Systems Research (CIDR '22). January 9-12, 2022, Chaminade, USA.

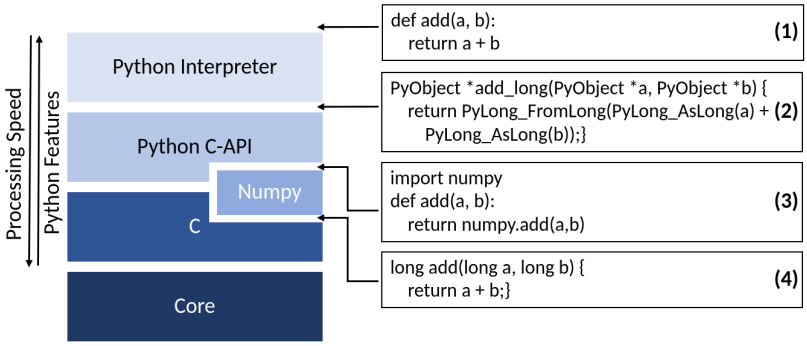

Figure 2: Levels of executing Python code

In this work, we elaborate possibilities to move down the execution stack shown in Figure 2 for scalar Python UDFs, while also evaluating the following orthogonal features for performance improvement:

- **Compilation:** We evaluate different compilation frameworks to transparently convert Python code into lower-level code. Additionally, we show how to dynamically use compiled code during database runtime, which is a challenging task in interpreted query engines, and provide an open-source prototype engine for evaluation.
- **Vectorization:** We elaborate on the impact of Python code vectorization to reduce call overhead.
- **Parallelisation:** We investigate how to parallelise Python code in a parallel query engine, which requires mitigating the global interpreter lock (GIL) issue.

2 RELATED WORK

Python transpilers like Cython [1] and Nuitka[1] work with arbitrary Python language constructs and especially also with external modules, but show slower performance compared to native code as they still rely on Python's C-API. Python compilers like Numba [7] translate Python code to native C code, which leads to efficient code execution but limits the usage to the supported language constructs. Numba supports a set of basic Python language constructs as well as Numpy code and offers an "object mode" as a fallback for working with, e.g., Strings, which also invokes the Python environment. However, importing external libraries is not allowed in Numba. Tuplex [15] focuses on compiling data analytics pipelines written in Python to a native binary. The framework is not publicly available at the time of writing. All these frameworks focus on compiling Python code to a binary and running it as a standalone process. In our work, we investigate how these frameworks can be integrated into an interpreted query engine during runtime, enabling the ad-hoc creation and usage of compiled Python UDFs. Besides these frameworks, GraalVM[2] supports compiling code into native images. However, it mainly focuses on Java code and only supports Python as a "guest language" embedded into Java code for compilation. The feature is in an experimental state and only supports a few Python modules. Several approaches also compile UDFs that are not written in Python. [3] focuses on compiling interpreted PL/SQL UDFs into recursive SQL functions to avoid PL/SQL - SQL context switches, while [14] aims at just-in-time compiling SQL lambda functions.

The vectorized, column-oriented database system MonetDB also supports Python UDFs [13]. It uses the interpretation of Python code as the standard case, but optimizes for Numpy UDFs as Numpy and column-stores share a similar storage layout. As a result, the execution of Numpy UDFs reaches nearly native code speed. [13] also highlights the issue that compiling Python UDFs would require to compile and link the database engine against the compiled UDF, which is not possible during database runtime. In our approach we focus on arbitrary Python constructs and exploit dynamic loading to make compiled UDFs available to the query engine during runtime.

In Python there is the global interpreter lock[3] (GIL), which was chosen in the design of Python in order to simplify low-level details like synchronisation or memory management of concurrent threads. While this leads to high single threaded performance, it limits the parallel or multithreaded execution of Python code [10]. Several solutions on the GIL issue in dynamic languages have been proposed like hardware transactional memory [12], software transactional memory [10] or the use of virtual machines [11]. In our approach, we achieve parallelism similary to virtual machines by spawning separate UDF engine processes in a parallel query engine. Working on independent data partitions, we do not need any type of synchronisation, which simplifies the GIL issue and makes separate Python processes a reasonable solution.

3 COMPILATION

As shown in Section 2, several frameworks target at increasing Python code performance by compiling Python code just-in-time or ahead-of-time. Similar to this approach of interpreted vs. compiled Python code, the dominant database design approaches of the last decades are also interpreted (vectorized) query engines vs. compiled query engines. While interpreted query engines built a query execution plan from a set of pre-defined operators and follow an iterator model like the Volcano iterator model [4], compiled query engines produce code for each specific query and compile it to an executable. In comparison to interpreted engines, there is a trade-off between the additional effort to produce and compile code for each query and the faster execution of optimized and compiled code. While using just-in-time compiled UDF code in queries of a compiled query engine involves linking of the query binary and is therefore straight forward, using compiled code in an interpreted

[1] http://nuitka.net/
[2] https://www.graalvm.org/
[3] https://wiki.python.org/moin/GlobalInterpreterLock

query engine during runtime is more challenging. In this section, we investigate Python code compilation and integration by presenting a prototype engine that offers compiled UDF operators, which realize compilation, dynamic loading and UDF execution. The design of these operators considers the following challenges:

- **C1 Transparency:** From a user perspective, there is no additional effort necessary to exploit compiled UDFs.
- **C2 External code:** Importing external libraries in the UDFs must be possible to benefit from the wide range of Python frameworks.
- **C3 Typing:** The operator needs to fill the gap between the strictly-typed database system and the weakly-typed Python code.
- **C4 Safety:** The system must be secured from unauthorized access over Python UDFs.

3.1 Engine design

Our prototype engine[4] follows the common properties of modern analytical database systems and is written in C. It is designed as an in-memory engine that organizes data in a columnar layout. Column values are tightly packed as arrays in memory resulting in efficient access when iterating over this data. Variable-sized types like strings are stored as pointers to the memory region containing the data. Additionally, the engine follows the Volcano iterator model [4], so an operator implements a common interface of an `open()`, `next()` and `close()` function. The `next()` functions are implemented using vector-at-a-time processing. As the intention of the prototype is to explore the design space of UDFs, it only contains a basic file scan operator, multiple different UDF operators, logic to "consume" the query result at the end of a query and a fine-grained profiling mechanism.

We integrated the most popular compilation/transpilation frameworks Numba, Nuitka and Cython into our prototype engine and created a separate operator encapsulating each of the frameworks. During the `open()` call, the UDF code is compiled using the respective framework, dynamically loaded into the running engine and initialised. As a baseline, we also designed a native Python UDF operator, which is based on the Python C-API[5]. In order to achieve the transparency goal C1, the native Python operator can be used as a fallback in case of compilation or loading errors.

3.2 UDF compilation

The UDF compilation step is similar for each of the used compilation frameworks and only differs in minor, framework-specific details. However, all frameworks also share the same challenge related to the way Python handles imported modules. All frameworks are originally designed to produce a compiled module that is used in the Python interpreter again. When facing an `import` statement, the Python interpreter calls the module's initialisation method which initialises the module and provides an entry point to the module's functions. A subsequent function call is then resolved using this entry point. Consequently, a module's functions are not visible to code outside the module, especially not for our query engine which does not run in a Python context.

In order to make the UDF code visible, one could manually rewrite the generated code. However, this is fundamentally different for different frameworks and makes the compilation process highly dependent on the compilation framework version, as each change in the code generation would require changes in the manual code rewriting. The design of the compilation step follows the goal of abstracting as much as possible from the used framework. Therefore, we solve this issue using a rather technical trick. For the compilation, we configure the compilation frameworks to keep intermediate states of the compiled code. This way, we get access to the object files in the Executable and Linkable Format (ELF) before they are linked together to a shared library. ELF is a standard format for shared libraries, objects or executables and has a fixed structure in its header and data segments, being a flexible representation for code and data. As part of the ELF format, symbol tables are used to maintain symbolic references, e.g. functions, to the respective part in the code segment. Function symbols are visible outside of a library or executable when they belong to the dynamic symbol table, having global visibility and default binding properties. Using ELF manipulation tools like LIEF[6] we manipulate the visibility and binding of the function symbol that belongs to the UDF accordingly and repeat the last compilation step to produce the shared library. Consequently, we chose to accept duplicate work in favor of abstraction from the compilation framework, so we do not change a framework's compilation behaviour to manipulate the generated files in between, but simply repeat the last linking step.

In the compilation step, we also make the first step to solve challenge C3. When the UDF is created, the user needs to specify input and output data types of the function. This is not a restriction, as this metadata is also available in a later integration into a full-fledged database system, where a user needs to specify the types in a `CREATE FUNCTION` query. Using this metadata we annotate the UDF code when necessary, like in the case of Numba. Additionally, supporting external modules as described in challenge C2 is a property of the compilation framework and therefore automatically provided, except for the Numba framework which only supports the Numpy library as an import. In order to achieve safety as stated in challenge C4, we maintain a banlist of modules and scan the UDF code before the compilation for the items of the banlist. This way, we avoid that users can execute statements potentially dangerous for the system or the underlying environment, e.g. by using the `os` or the `sys` module.

The compilation process is performed using a Python script consisting of calling the framework, modifying symbol visibility and re-running the last compilation step. As this script is not changed during database runtime and consists of Python code, it can be compiled using the compilation script itself to produce a shared library containing the compilation logic and dynamically linked with the database engine. As a result, calling library functions is faster than invoking a Python script out of the database engine.

3.3 Dynamic loading and initialisation

After compilation, the library containing the UDF must be stored at a well-defined place that is accessible for the system. While this can be easily achieved in single-server database systems, we argue

[4] https://github.com/dbis-ilm/Compiled_UDF_engine
[5] https://docs.python.org/3.6/c-api/
[6] https://github.com/lief-project/LIEF

```
typedef void (*fplaceholder)(void);
typedef void (*WrapperFunc)(fplaceholder,
    char**, char**);
// For every parameter combination
typedef double (*dld)(long, double);
void func_38(fplaceholder f, char **params,
     char **out) {
  long* p0   = (long*) &params[0];
  double* p1 = (double*) &params[1];
  *((double*)out) = ((dld)f)(*p0, *p1);}
```

Listing 1: Example of the wrapping code

that even for distributed database systems this is not a limitation due to the presence of cloud storage in the cloud environment or shared storage like HDFS in the case of on-premise cluster solutions. In the next step, the library is now dynamically loaded into the running database application using `dlopen/dlsym`[7]. The loading step is thereby driven by the goal to avoid as much effort as possible in the later execution, especially expensive branching.

Cython, Nuitka and Numba's object mode are based on Python's `PyObjects`, which are the type abstraction of Python's C-API and can contain arbitrary types. For these frameworks, we initialise input and output conversion functions based on the UDF signature and store them as function pointers to avoid branching during execution. The actual UDF's signature may however consist of arbitrary parameter type combinations. As this is not known during the compilation of the database engine, we define a placeholder function type and generate a wrapper function and a function definition for every possible UDF signature. In practical cases, the number of possible UDF parameters must be reasonably limited. The wrapper is correctly set according to the UDF signature during the loading step and therefore introduces branching only once. As shown in an example function for a UDF that takes a long and a double parameter and returns a double in Listing 1, this wrapper casts the generic input parameters as well as the actual UDF and calls the UDF. The result is then stored at the given buffer location, which is a part of the result vector. By storing the wrapper code as a function pointer, we avoid branching during the UDF execution while also solving the typing challenge C3.

Besides the UDF, we also load the symbol for the library's initialisation function if required by the compilation framework. Again for Cython, Nuitka and Numba's object mode, we initialise the module in the Python enviroment afterwards potentially using Python's multi-phase initialisation[8]. Note that this might happen after the Python interpreter was initialised during the engine startup, depending on a system's behaviour to use the Python interpreter, i.e. long-running vs. starting a (sub-)interpreter per query. This initialisation enables access to the module's internal data structures and helper functions, being the equivalent of Python's `import` statement.

3.4 UDF execution

As described in Section 3.1, our prototype engine follows the Volcano iterator model in a vector-at-a-time fashion and has a columnar storage layout. Consequently, each UDF operator receives a set of vectors (one for each column) from the child operator in the query tree. Keeping these vectors unchanged, the operator adds a result vector for the UDF output to the intermediate result. If required by the used compilation framework, the UDF input values are converted to `Py_Object` elements using the respective input conversion functions. The compiled UDF is called using the wrapper function and the results are converted back using the output conversion functions if necessary. These three steps are performed in a vectorized fashion. Note that there is no branching in this step, as function pointers were correctly set during the dynamic loading and initialisation as described in Section 3.3.

Another important aspect of an execution engine is NULL handling. In our approach, we basically have two options where to perform NULL handling: in the database engine or in the UDF code. The engine-internal handling would require to only execute the UDF on non-NULL values. In the vectorized variant, one would therefore need to track the offsets inside the vectors to be able to put results into the right position again after UDF execution. Performing the NULL handling in the UDF on the other hand simply requires the input conversions to produce `Py_None` as Python's NULL indicator. In our engine prototype, we decided for the second option to perform NULL handling in the UDF, as this offers more flexibility to the user to handle NULLs. One could imagine use cases where UDFs explicitly need to handle cases where some parameters are NULL, which would not be possible to support when UDF calls are skipped by the engine-internal NULL handling.

4 VECTORIZATION

The performance of the UDF execution is highly dependent on the way the UDF code is written. In commercial systems, user-written UDF code is likely to get wrapped into a code template. This way, customised error checking mechanisms, type conversion and interpretation for complex types and vector iteration mechanisms can be achieved. One can think about a user function that is called from a main function in different ways. If the user code is written in a tuple-at-a-time fashion, the user function is called for each element of the vector, introducing significant call overhead. If the user code is written in a vector-at-a-time fashion, the user function is only called once, significantly increasing speed. During our design we found the call overhead inside the Python code as one of the most limiting factor for performance. Using a vectorized UDF that preallocates the result vector and uses list comprehension (potentially zipping multiple input vectors before) showed the best performance.

In the current project state UDF vectorization is enabled manually by the user by (1) rewriting the UDF code to the vectorized version and (2) indicating vectorization using a prefix in the UDF naming. In order to further extend usability and reach the transparency goal C1, we aim at an automatic UDF rewriting mechanism in the future that embeds the UDF code into a vectorization template.

5 PARALLELISATION

Using UDF compilation as described in the Section 3 is intended to increase single-core performance of the UDF execution in a vectorized, interpreted query engine. However, today's analytical database systems are typically massively parallel processing (MPP)

[7] https://linux.die.net/man/3/dlsym
[8] https://www.python.org/dev/peps/pep-0489/

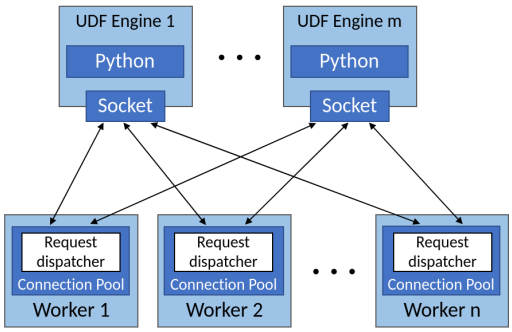

Figure 3: Overview over the UDF engine pool

systems to benefit from the increasing number of cores per socket and the possibility to scale systems in cluster environments. In this Section we want to describe possibilities to also scale the UDF execution, which is orthogonal to the compilation approach.

We assume a MPP system that runs multiple workers to execute query plans or subplans in parallel on independent data partitions. When integrating our compiled UDF approach in such a system, the frameworks that still use the Python environment (Cython, Nuitka, Numba object mode) would share the Python interpreter. As described in Section 2, this case would lead to high contention due to the GIL issue. Consequently, enabling parallelisation in Python leads to mitigating the GIL issue. Instead of using special transactional memory, we designed UDF engines that run as processes separately from the actual database workers as sketched in Figure 3 (Note: workers might be either processes or threads, but UDF engines are separate processes). These UDF engines start their own Python environments, which are therefore isolated by the process semantics. Database workers and UDF engines can be scaled independently and are connected over point-to-point socket connections. On the database engine side, these connections are maintained in a connection pool and requests (i.e. UDF execution requests) are dispatched over a scheduling strategy. For now, we use a simple round robin assignment strategy to assign requests to UDF engines. As vector sizes are fixed, this simple strategy already provides a balanced utilization of the UDF engines. Even when data is skewed, the unbalanced load in the database workers would be balanced among the UDF engines. After execution, results are sent back to the database workers over the socket connection. This "remote UDF" approach is again a tradeoff, as it introduces inter-process communication and data transfer while enabling parallel execution in the Python environment. As a side effect, moving the Python execution to separate processes offers possibilities to improve the security of running Python, e.g. by running the UDF engines in containers. Sandboxing Python execution is a complex problem and the reason for Python UDFs being only available as a beta version or requiring administrator rights e.g. in Postgres[9]. It is therefore not the scope of this paper.

6 EVALUATION

In the evaluation, we compare the code generated by the integrated frameworks as well as the impact of compilation, vectorization and parallelisation. The evaluation was performed on a machine

[9]https://www.postgresql.org/docs/13/plpython.html

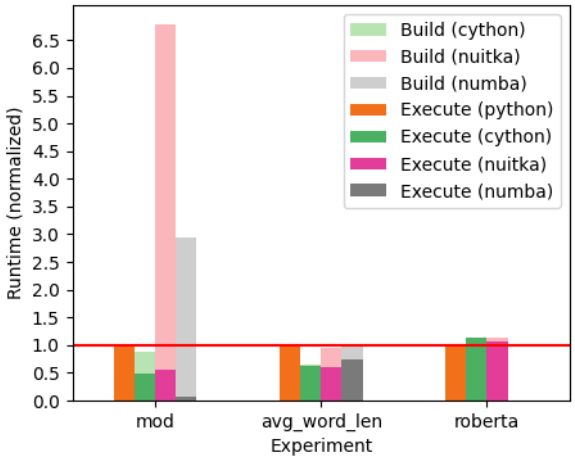

Figure 4: Compilation framework comparison in prototype engine

consisting of a Intel(R) Core(TM) i7-8565U CPU @ 1.80GHz and 24 GB RAM. We designed a set of experiments to represent different use cases:

- **Mod:** A simple modulo UDF, representing a mathematical operation.
- **Avg_word_len:** Splitting a string and computing the average word length, representing a string analytics function that consumes the whole string.
- **Roberta:** Using the Roberta ONNX model [8] to analyse the sentiment of texts.

The experiments were performed on the TPC-H [2] partsupp table at SF10 and the IMDB movie review dataset [9]. In the first experiment we compare the behaviour of the integrated compilation frameworks using the prototype query engine. In this experiment, we manually write wrapped UDF code as explained in Section 4 in order to simulate the behaviour after database system integration. Figure 4 shows the results for the described experiments with the runtime being normalized against the native Python operator runtime, which acts as the reference. Build runtime including compilation is in a similar order of magnitude for each experiment, however taking a different part of the overall runtime depending on the execution runtime. For the modulo experiment we can observe a performance benefit for all compilation frameworks, with Numba being the best one as it produces native C code in this example equivalent to the lower baseline. It reaches stack level (4) in Figure 2 and runs around 20 times faster than the reference. For string analytics in the avg_word_len example, we can again see a performance benefit for all frameworks. However, Numba is slower than Nuitka and Cython here as it falls back to the Python object mode to handle strings. In the Roberta example, Numba is not applicable as the UDF uses an imported module. As the compilation frameworks obviously cannot influence the module code and the ONNX runtime environment, which is responsible for the major part of the runtime in this example, compilation does not lead to a significant benefit here. As we aim at a solution that is transparently applicable for all use cases, Numba is not an option. Nuitka and Cython both reach stack level (2) of Figure 2 and showed similar performance in all cases leading to a speedup of around 2

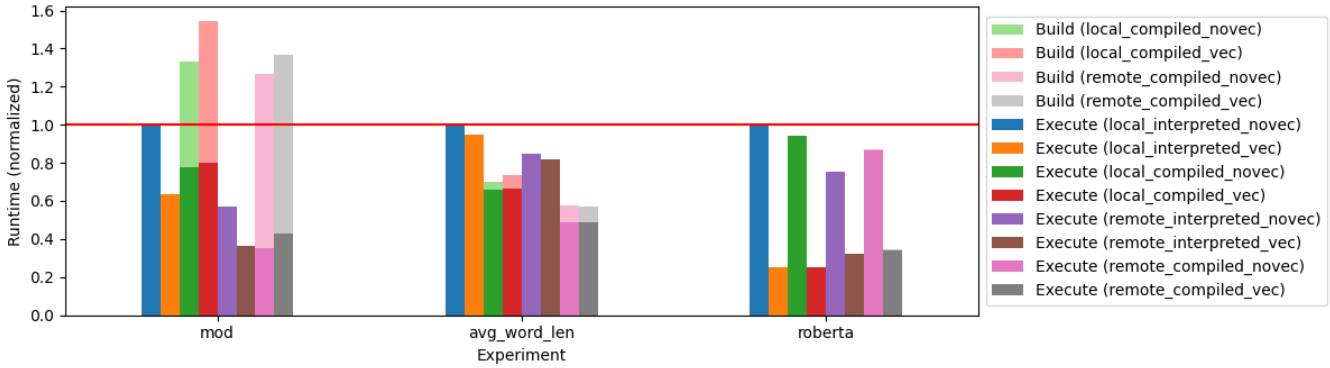

Figure 5: Actian Vector results

in mathematical operations and string analytics, while Cython has significantly lower compilation effort.

Based on the first experiment, we integrated the UDF approach using Cython as the compilation framework into the Actian Vector database system at version 6.1. In the second experiment we therefore operate on stack level (2) of Figure 2 and evaluate end-to-end query performance using combinations of compilation, vectorization and parallelisation. The queries thereby consist of a scan, the UDF and the aggregation of the result to reduce the result size. The runtime results are shown in Figure 5, again normalized against the reference runtime, which is the interpreted, non-vectorized and non-parallelised version. We can observe that compilation has a significant benefit compared to the respective interpreted variants, with compilation expectedly being only slightly better than interpretation in the Roberta example, as major part of the runtime takes place in the ONNX runtime environment. Vectorization also leads to performance gains in nearly every case except the compiled variants, as the Cython framework seems to have problems to produce efficient code for list comprehension. Cython provides an advanced Python dialect, offering more efficient compilation for specially annotated code constructs. This could be exploited in the future by automatically rewriting the user defined python code into the Cython dialect in order to produce faster compiled code. However, Roberta especially benefits from vectorization, as the ONNX model needs to be loaded only once per vector instead once per tuple. Parallelisation using a parallelisation degree of 4 remote UDF engine processes additionally increases performance in our experiments except the Roberta example, as this experiment relies on the scalability of the ONNX runtime. Overall we can derive the following key findings from our evaluation:

- Only stack level (2) of Figure 2 is reachable when transparently supporting arbitrary python code.
- Compilation together with parallelisation leads to the best performance in most experiments with speedups between 2 and 3 compared to the Python execution.
- Compilation does not have an impact on functions that rely on external modules.
- Parallelisation using external UDF processes introduces overhead of inter-process data exchange, but leads to a speedup during execution by mitigating the GIL issue.
- Vectorization leads to performance gains by reducing Python call overhead.

Consequently, these features are a good choice for a broad range of use cases like user-specific calculations, scoring functions, analytics or ML pre- or post-processing steps.

7 CONCLUSION

User-defined Python functions are easy-to-use extensions to database engines and provide support for modern analytical tasks. Motivated by the fact that the execution of Python code suffers from bad performance and scalability, we explored the design space of accelerating Python UDFs in vectorized query engines. We evaluated the impact of vectorization, parallelisation and compilation, with the latter being realized by different existing compilation frameworks. We show how UDFs can be compiled just-in-time during database runtime, dynamically loaded and used in query execution and implemented this feature as an open-source prototype engine. Furthermore we presented an approach to mitigate Python's GIL issue using separate UDF engine processes to enable parallel UDF execution. In our evaluation, we integrated the Cython compilation framework into the Actian Vector database system and evaluated end-to-end query performance for representative use cases. We thereby achieved significant speedups for different use cases with compilation and parallelisation used together. With transparency as a main design goal, the presented approaches are easy-to-use and work for arbitrary Python code constructs. In the future, we aim at evaluating automatic code rewriting to exploit compilation framework specific characteristics and annotations. Machine learning workloads still rely on the performance of external runtime environments, which could be mitigated by a more native integration of ML models.

REFERENCES

[1] S. Behnel et al. Cython: The Best of Both Worlds. *Computing in Science & Engineering*, 13, 2011.
[2] P. A. Boncz, T. Neumann, and O. Erling. TPC-H Analyzed: Hidden Messages and Lessons Learned from an Influential Benchmark. In *TPCTC*, volume 8391, pages 61–76. 2014.
[3] C. Duta, D. Hirn, and T. Grust. Compiling PL/SQL away. In *CIDR*, 2020.
[4] G. Graefe. Volcano— An Extensible and Parallel Query Evaluation System. *IEEE Trans. on Knowl. and Data Eng.*, 6(1):120–135, 1994.
[5] S. Gupta and K. Ramachandra. Procedural Extensions of SQL: Understanding their usage in the wild. *PVLDB*, 14(8):14, 2021.
[6] S. Kläbe and S. Hagedorn. Applying machine learning models to scalable dataframes with grizzly. In *BTW*, pages 195–214, 2021.
[7] S. K. Lam, A. Pitrou, and S. Seibert. Numba: A LLVM-Based Python JIT Compiler. In *LLVM*, 2015.

[8] Y. Liu et al. RoBERTa: A Robustly Optimized BERT Pretraining Approach, 2019. _eprint: 1907.11692.
[9] A. L. Maas and others. Learning Word Vectors for Sentiment Analysis. In *ACL-HLT*, pages 142–150, 2011.
[10] R. Meier and A. Rigo. A way forward in parallelising dynamic languages. In *ICOOOLPS*, pages 1–4, Uppsala, Sweden, 2014.
[11] R. Meier, A. Rigo, and T. R. Gross. Virtual machine design for parallel dynamic programming languages. *PACMPL*, 2:1–25, 2018.
[12] R. Odaira, J. G. Castanos, and H. Tomari. Eliminating global interpreter locks in ruby through hardware transactional memory. In *PPoPP*, pages 131–142, Orlando, Florida, USA, 2014.
[13] M. Raasveldt and H. Mühleisen. Vectorized UDFs in Column-Stores. In *SSDBM*, pages 1–12, 2016.
[14] M. E. Schüle et al. Freedom for the SQL-Lambda: Just-in-Time-Compiling User-Injected Functions in PostgreSQL. In *SSDBM*, 2020.
[15] L. Spiegelberg et al. Tuplex: Data Science in Python at Native Code Speed. In *SIGMOD*, pages 1718–1731, 2021.