Demonstration of Udon: Line-by-line Debugging of User-Defined Functions in Data Workflows

Yicong Huang
Department of Computer Science
University of California, Irvine
Irvine, CA, USA
yicongh1@ics.uci.edu

Zuozhi Wang
Department of Computer Science
University of California, Irvine
Irvine, CA, USA
zuozhiw@ics.uci.edu

Chen Li
Department of Computer Science
University of California, Irvine
Irvine, CA, USA
chenli@ics.uci.edu

ABSTRACT

Many big data systems are written in languages such as C, C++, Java, and Scala for high efficiency, whereas data analysts often use Python to conduct data wrangling, statistical analysis, and machine learning. User-defined functions (UDFs) are commonly used in these systems to bridge the gap between the two ecosystems. Debugging complex UDFs in data-processing systems is challenging due to the required coordination between language debuggers and the data-processing engine, as well as the debugging overhead on large volumes of data. In this paper, we showcase Udon, a novel debugger to support line-by-line debugging of UDFs in data-processing systems. Udon encapsulates modern line-by-line debugging primitives, such as those to set breakpoints, perform code inspections, and make code modifications while executing a UDF on a single tuple. In this demonstration, we use real-world scenarios to showcase the experience of using Udon for line-by-line debugging of a UDF.

CCS CONCEPTS

• **Information systems** → **Data management systems**; • **Software and its engineering** → **Software testing and debugging**.

KEYWORDS

data workflows, user-defined functions, debugging, python udf

ACM Reference Format:
Yicong Huang, Zuozhi Wang, and Chen Li. 2024. Demonstration of Udon: Line-by-line Debugging of User-Defined Functions in Data Workflows. In *Companion of the 2024 International Conference on Management of Data (SIGMOD-Companion '24), June 9–15, 2024, Santiago, AA, Chile.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3626246.3654756

1 INTRODUCTION

Big data systems have become increasingly popular due to their capabilities to process large and complex datasets. A key feature of these systems is the integration of user-defined functions (UDFs), which allow users to implement custom logic for specific data-processing tasks. These UDFs enhance the systems' usability, especially for tasks where built-in functions are inadequate. UDFs facilitate the incorporation of third-party code and libraries, such as machine learning tools in Python and statistical packages in R, broadening the applicability of these systems. Given the flexibility of UDFs, they become increasingly complex. For instance, the workflow in Figure 1 includes UDF operators for sophisticated machine learning training and customized aggregations.

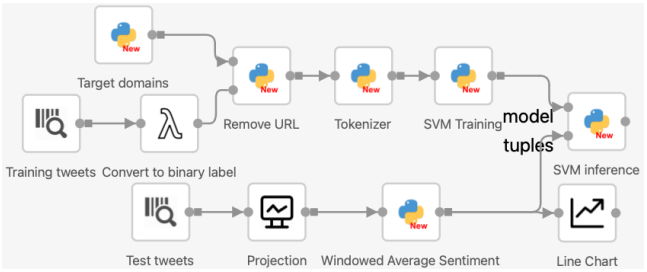

Figure 1: A data workflow with UDFs for SVM-based training and inference and for calculating aggregated sentiment.

Due to their increasing complexity, developing UDFs can pose significant challenges, especially when it comes to debugging. First, it requires coordination between the data-processing engine and the debugger to control the UDF execution. Second, debugging can introduce computational overhead to the UDF execution, which can be significant on large datasets or expensive UDFs. Traditional debugging methods often involve post-mortem log analysis, which can be inefficient and insufficient for identifying runtime errors, especially when dealing with large datasets and complex UDF operators. An alternative approach involves testing UDFs with sample data. This testing method may not cover all the scenarios, such as missing data or issues that arise only on the entire dataset.

In our recent work, we introduced Udon [7], a novel debugger for UDFs in data-processing systems. Udon offers an on-demand, line-by-line debugging experience while a data-processing task is actively running. It runs debuggers under the control of the data engine, and employs a unique debugging-aware execution model to support operator responsiveness during debugging. It incorporates various optimization techniques for minimizing runtime overhead by intelligently detaching debuggers when they are not required.

In this demonstration we will showcase Udon's capabilities through practical debugging scenarios from real-world applications. Given the popularity of Python UDFs, we integrated the standard Python debugger, pdb, into Udon and seamlessly integrated Udon into Texera [11], an open-source data analytics workflow system. We will highlight the following features of Udon:

(1) It allows attachment and detachment of language debuggers for UDFs that are running in a distributed environment.
(2) It offers a fine-grained line-by-line debugging experience for complex UDFs.
(3) It supports inspection and modification of intermediate states between code lines in a UDF.

SIGMOD-Companion '24, June 9–15, 2024, Santiago, AA, Chile

ACM ISBN 979-8-4007-0422-2/24/06
https://doi.org/10.1145/3626246.3654756

2 DEMONSTRATION SYSTEM OVERVIEW

Figure 2 shows the architecture of Udon. We develop Udon on top of Texera [11], an open-source system to support cloud-based collaborative data analytics using workflows. Texera provides a web interface where analysts can construct data analytical workflows using operators. In addition to the built-in operators, Texera supports Python UDF operators and allow users to edit the UDF code on operators. The workflow execution is distributed on a cluster of machines. To integrate Udon, we introduce a debugger frontend into the interface of Texera. On the execution engine, we utilize the debugger-enabled UDF operator and debug-aware coordinator.

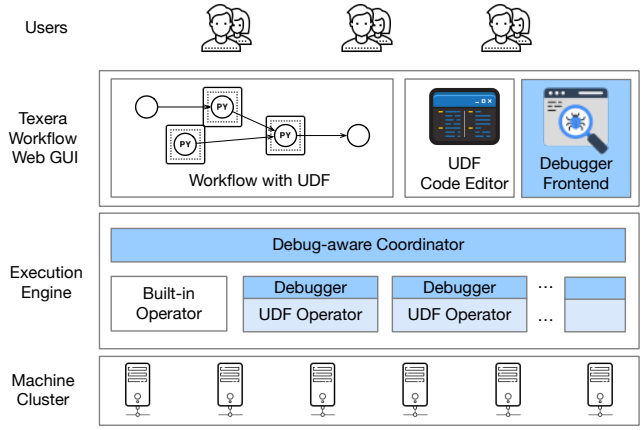

Figure 2: System architecture of Udon integrated into Texera. Udon components are highlighted in blue.

Engine-controlled language debuggers. Udon integrates the standard Python debugger, pdb [10]. Each UDF operator can be attached with a pdb instance to enter a debug mode. The debug commands from users are sent to the coordinator, then forwarded to the target debugee operator. The coordinator is aware of all operators in either the debug mode or the normal execution mode.

Two-thread UDF execution model. Udon uses a novel two-thread execution model for UDF operators to facilitate communication between the coordinator and the UDF operators. Each debugger-enabled UDF operator provided consists of two threads: a control-processing (CP) thread and a data-processing (DP) thread. These two threads communicate with each other through shared variables. The CP thread is responsible for receiving control instructions from the coordinator, such as system commands (e.g., heartbeat and statistic checks) and debug instructions (e.g., setting breakpoints). The DP thread is responsible for executing the UDF code to process data tuples, and it can be managed by the language debugger.

Udon uses optimizations to detach language debuggers based on specific data characteristics. In this demonstration we will enable the optimizations to mitigate the runtime overhead associated with the language debuggers. See the full paper [7] for a detailed description of these features.

3 DEMONSTRATION SCENARIOS

In this section we will walk through the demonstration of Udon with the help of Alice, an imaginary Texera user. Consider the scenario where Alice is developing a Python UDF using the Texera workflow interface, as shown in Figure 3. The goal is to calculate the windowed average sentiment of texts. The UDF processes one tuple at a time, each containing a text string. It begins by utilizing an NLTK's sentiment analyzer to compute a sentiment score for the tuple's text, and the score is between -1 and 1. It maintains a window to store recent sentiment scores, and uses this window to compute an average sentiment score of the window for each tuple.

```
1  from pytexera import *
2  from nltk.sentiment import SentimentIntensityAnalyzer
3  class WindowedSentimentOperator(UDFOperatorV2):
4
5    def open(self):
6      self.window_size = 30
7      self.sum = 0
8      self.sentiments = []
9      self.analyzer = SentimentIntensityAnalyzer()
10
11   def process_tuple(self, tuple: Tuple):
12     text = tuple['text']
13     polarity_scores = self.analyzer.polarity_scores(text)
14     sentiment_score = polarity_scores['compound']
15
16     # Update the sum and queue for sentiments
17     self.sum += sentiment_score
18     self.sentiments.append(sentiment_score)
19     if len(self.sentiments) > self.window_size:
20       expired_sentiment = self.sentiments.pop(0)
21       self.sum -= expired_sentiment
22
23     current_average = self.sum / len(self.sentiments)
24
25     # Add the average sentiment to the tuple for output
26     tuple['windowed_average'] = current_average
27     yield tuple
```

Figure 3: A UDF by a user Alice to calculate the windowed average sentiment of texts.

Alice has tested the code on sample data. When she incorporates the UDF into a production workflow shown in Figure 1 to generate a line chart displaying changes in sentiment, the output shows an average sentiment of 0 for several consecutive tuples. An average sentiment score is expected to be a fraction number, but during the execution of the workflow, the operator always generates a score of 0. This unexpected behavior suggests the execution has bugs. Alice's code may not contain a bug, but the issue could be from third-party functions or due to incorrect input data. In this case, Alice plans to use Udon line-by-line debug the issue.

3.1 Suspending a running UDF operator between two code lines

Alice wants to trace the execution of the UDF code line by line to gain insights of the runtime behavior. To do so, she can suspend the UDF operator at the line where the issue appears.

Setting a line breakpoint in UDF code. One way to suspend the UDF at a desired line is to set a line breakpoint. To do so, Alice clicks on the line 26 to set a breakpoint, with a condition $current\_average == 0$. This condition allows her to capture the output tuple matching this condition for further analysis. This step sends the following debug instruction to the coordinator

$$DebugCommand(SetBreakpoint, 26, \text{“}current\_average == 0\text{”}),$$

which is sent to the target debuggee operator. When the operator receives a `SetBreakpoint` command, the CP thread updates a shared variable. The DP thread has a chance between the execution of two lines of UDF code to explicitly check whether there is a debug command to handle. If so, it attaches the language debugger to itself and hands over its execution control to the debugger, which then registers the breakpoint and starts to monitor it. In this example, each yield statement (e.g., at line 27) will allow the DP thread to conduct an explicit check, and Alice can use `yield` statements in the code to increase this frequency.

Waiting for a breakpoint hit. After the breakpoint is set, Alice waits till the UDF execution reaches the breakpoint. During the execution of the DP thread, the language debugger will check if any breakpoint is hit before executing the next code line. When there is a hit, the language debugger writes the hit event to another shared variable, suspends the DP thread's execution, notifies the CP thread, and awaits further debug instructions. The CP thread then forwards the event to the coordinator and eventually back to the user on the frontend. The snapshot in Figure 4 is taken after the operator has reached the breakpoint at line 26.

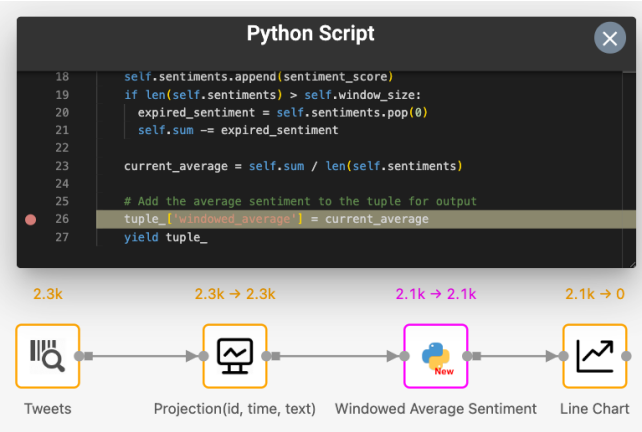

Figure 4: A breakpoint is hit at line 26, as indicated by the highlighted code in the editor. The operator is also marked in purple, indicating that it has been suspended.

During this suspension, the operator is highlighted in purple, which shows its paused state. The other operators remain in an active state, shown by the orange color. As the debugee operator ceases to emit data, downstream operators will complete processing their input and await further input. Similarly, because the debugee operator cannot handle additional input data, upstream operators experience back pressure at the network layer, which prevents them from sending data to the debuggee operator. Consequently, the workflow gradually enters a state of a full pause.

Catching an exception during processing of a tuple. An alternative method to suspend the execution involves capturing a runtime exception raised within the UDF code. Developers often introduce additional assertions and validations in the code to scrutinize edge cases. For instance, Alice can insert an assertion $assert(current\_average! = 0)$ after line 23 to validate the code. When the assertion fails, the UDF will throw an exception at this line, and the debugger will catch and report it as a debug event, which is similar to a breakpoint hit. Consequently, the DP thread is suspended by the debugger, awaiting further debug instructions.

3.2 Line-by-line Execution Control in UDF

Once the execution is suspended, Alice has complete control of the UDF execution, and can closely monitor the UDF line by line.

Inspecting intermediate states between UDF lines. Alice can inspect the states of a UDF between code lines. For example, when the execution is suspended at line 26, she can print the value of "text" by sending a $print(text)$ debug command. She can also evaluate the predicate $len(self.sentiments) > 0$ by entering the predicate as a debug command to make sure the division is on a non-positive number. In addition, she can also set watchpoints to follow the update of any intermediate states. For example, she can send a $display\ self.sum$ debug command and suspend the execution at any code line where the variable is updated.

Stepping within UDF code during the processing of a tuple. Alice can control the UDF execution to the next over, the next function, the next breakpoint, or even the next tuple at the same code line. This functionality allows her to examine the intermediate states at different lines. In the provided example, the $sentiment\_score$ at line 14 is generated by a library call within NLTK. Alice may find it beneficial to step into the $polarity\_scores$ function and have the insight of the third-party library.

Retrying the processing of the current tuple. To facilitate the debugging of the tuple that causes the issue, Udon supports repeated execution of the tuple. Alice can initiate this by clicking the "Retry Tuple" button, which resubmits the tuple for another execution of the UDF code. This functionality is valuable when Alice wants to backtrack to a previous execution point for the same tuple. In the provided example, Alice temporarily suspends the UDF execution at line 26 but wants to inspect the function call located at an earlier line 14. To achieve this, she can utilize the "Retry Tuple" feature, which discontinues the UDF execution for the current tuple and repeats the UDF for the same tuple.

3.3 In-place correction of a UDF operator

The underlying issue stems from two possible reasons: 1) *Data errors*, which are incorrect values in the input tuple; and 2) *Code errors*, which are logic flaws in the UDF code. After identifying a reason, a Udon user can fix UDF executions in-place, without changing other operators or disrupting the workflow's execution. In particular, Alice can take the following actions to fix them.

Fixing a data error within a UDF operator. If an input tuple contains incorrect values, Alice can correct these values within the UDF operator during execution, between code lines. Suppose Alice identifies that a specific tuple contains special characters encoded in a manner incompatible with $NLTK.SentimentIntensityAnalyzer$. In this case, she can send a $tuple[\text{"}text\text{"}] = \langle new\_text \rangle$ debug command to fix the tuple in-place before executing line 12, overwriting the original text at the beginning of the UDF. When continued, line 13 can invoke the analyzer to process the modified text.

Updating the UDF code during UDF execution. In cases where the UDF code logic requires a fix, Udon allows for in-place updates of the UDF code without terminating the operator or the workflow. Alice simply needs to provide an updated UDF code snippet through a debug command. The new code is compiled, loaded, and replaces the original flawed UDF logic. Any states of the old UDF

are transferred into the new UDF, and Alice has the flexibility to modify the states to align with the new code logic. Afterward, she can utilize the Retry Tuple feature to process the current tuple with the updated UDF and resume the workflow right from that point.

Debugging often requires trial and error, and developers may not always be certain about a fix. For example, Alice may have a hypothesis she wants to verify, or she might have a code change that she wants to test before committing the change. In such cases, Udon allows her to modify intermediate states within a UDF to evaluate "what if" scenarios. Suppose when the UDF execution is suspended at line 23, Alice wants to check how UDF executes if the $self.sum$ value is 0. She can change the intermediate state at line 23 by sending $self.sum = \langle testing\_value \rangle$ to the debugger (shown in Figure 5). This statement updates the intermediate state at line 23, enabling the execution to proceed with the testing value. Alice can always retry the execution of the same tuple with different intermediate states before making the final correction.

Figure 5: Alice uses the debugger frontend to assign a testing value to change intermediate states between lines.

3.4 Detaching the debugger from a UDF

After completing the debugging process, Alice can remove all breakpoints and watchpoints using the *clear* debug command. Subsequently, she can resume the normal UDF execution by sending the *continue* command, which tells the DP thread to detach the language debugger. Udon provides optimizations to minimize the overhead introduced by language debuggers, and this approach removes the debugger overhead once the debugging has finished.

3.5 Collaborative UDF debugging

Udon also supports collaborative UDF debugging, making it the first system to our best knowledge to provide this capability. Suppose Alice finds the SVM training operator has an unexpected behavior, but she has limited experience in ML. She can use the collaborative features [8] provided by Texera to invite her colleague, Bob, to help her debug the SVM operator. The two users share the same execution environment, and each of them can send debug commands to any UDF. For instance, Alice can set breakpoints in the SVM operator, suspend the execution to show the issue to Bob, and Bob can take control of the UDF execution to debug it together with Alice. Alice can monitor the debug commands executed by Bob and observe all the debug events generated by the UDF. If desired, Alice can delegate the entire debugging task for the SVM operator to Bob and debug the Windowed Average Sentiment operator on the same workflow execution. Since the two target operators are independent from each other, Alice and Bob will not interfere with each other's debugging efforts.

4 RELATED WORK

Big data engines such as Apache Spark and Apache Flink support Python UDFs and offer limited ability to use external debuggers [2, 3]. These solutions do not support coordination between the engine and the debugger, thus can cause the data tasks to fail unexpectedly during debugging. A recent survey [4] discussed a taxonomy of methods for integrating UDFs into data engines, and most approaches focus on optimization of UDF performance over debugging support. The existing debuggers on big data systems such as BigDebug [5] and TagSniff [1] focus more on the input and output data of each operator, not than the intermediate states of UDFs. They treat operators as black boxes and do not support line-by-line debugging inside a UDF. Other efforts [6, 9] involve transferring the debugging session as well as the data to an external process, which allows execution with a local debugger. These methods are not applicable if a bug only occurs in the original environment but not in a development environment. In our previous works, we showcased the pause feature [12] and the collaborative features [8] of Texera such as shared-editing and shared-execution of a workflow. This demonstration takes Texera system to the next level by introducing collaborative line-by-line UDF debugging.

ACKNOWLEDGMENTS

This work was funded by the National Science Foundation (NSF) under award III-2107150. We thank the Texera team at UC Irvine for their contributions to the development of the system.

REFERENCES

[1] Bertty Contreras-Rojas, Jorge-Arnulfo Quiané-Ruiz, Zoi Kaoudi, and Saravanan Thirumuruganathan. 2019. TagSniff: Simplified Big Data Debugging for Dataflow Jobs. In *Proceedings of the ACM Symposium on Cloud Computing, SoCC 2019, Santa Cruz, CA, USA, November 20-23, 2019*. ACM, 453–464. https://doi.org/10/mrgh

[2] Debugging | Apache Flink 2024. https://nightlies.apache.org/flink/flink-docs-master/docs/dev/python/debugging/.

[3] Debugging PySpark – PySpark 3.1.1 documantation 2024. https://spark.apache.org/docs/3.1.1/api/python/development/debugging.html.

[4] Yannis Foufoulas and Alkis Simitsis. 2023. User-Defined Functions in Modern Data Engines. In *39th IEEE International Conference on Data Engineering, ICDE 2023, Anaheim, CA, USA, April 3-7, 2023*. IEEE, 3593–3598. https://doi.org/10/mrgd

[5] Muhammad Ali Gulzar, Matteo Interlandi, Seunghyun Yoo, Sai Deep Tetali, Tyson Condie, Todd D. Millstein, and Miryung Kim. 2016. BigDebug: debugging primitives for interactive big data processing in spark. In *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*, Laura K. Dillon, Willem Visser, and Laurie A. Williams (Eds.). ACM, 784–795. https://doi.org/10.1145/2884781.2884813

[6] Pedro Holanda, Mark Raasveldt, and Martin L. Kersten. 2017. Don't Keep My UDFs Hostage - Exporting UDFs For Debugging Purposes. In *XXXII Simpósio Brasileiro de Banco de Dados - Short Papers, Uberlandia, MG, Brazil, October 4-7, 2017*, Carmem S. Hara, Bernadette Farias Lóscio, and Damires Yluska de Souza Fernandes (Eds.). SBC, 246–251. http://sbbd.org.br/2017/wp-content/uploads/sites/3/2018/02/p246-251.pdf

[7] Yicong Huang, Zuozhi Wang, and Chen Li. 2023. Udon: Efficient Debugging of User-Defined Functions in Big Data Systems with Line-by-Line Control. *Proc. ACM Manag. Data* 1, 4 (2023), 225:1–225:26. https://doi.org/10.1145/3626712

[8] Xiaozhen Liu, Zuozhi Wang, Shengquan Ni, Sadeem Alsudais, Yicong Huang, Avinash Kumar, and Chen Li. 2022. Demonstration of Collaborative and Interactive Workflow-Based Data Analytics in Texera. *Proc. VLDB Endow.* 15, 12 (2022), 3738–3741. https://doi.org/10.14778/3554821.3554888

[9] Matteo Marra, Guillermo Polito, and Elisa Gonzalez Boix. 2020. A debugging approach for live Big Data applications. *Sci. Comput. Program.* 194 (2020), 102460. https://doi.org/10.1016/j.scico.2020.102460

[10] pdb — The Python Debugger 2024. https://docs.python.org/3/library/pdb.html.

[11] Texera 2024. Collaborative Data Analytics Using Workflows, https://github.com/Texera/texera/.

[12] Zuozhi Wang, Avinash Kumar, Shengquan Ni, and Chen Li. 2020. Demonstration of Interactive Runtime Debugging of Distributed Dataflows in Texera. *Proc. VLDB Endow.* 13, 12 (2020), 2953–2956. https://doi.org/10.14778/3415478.3415517