Udon: Efficient Debugging of User-Defined Functions in Big Data Systems with Line-by-Line Control

YICONG HUANG, University of California, Irvine, USA
ZUOZHI WANG, University of California, Irvine, USA
CHEN LI, University of California, Irvine, USA

Many big data systems are written in languages such as C, C++, Java, and Scala to process large amounts of data efficiently, while 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. In this paper, we propose Udon, a novel debugger to support fine-grained debugging of UDFs. Udon encapsulates the modern line-by-line debugging primitives, such as the ability to set breakpoints, perform code inspections, and make code modifications while executing a UDF on a single tuple. It includes a novel debug-aware UDF execution model to ensure the responsiveness of the operator during debugging. It utilizes advanced state-transfer techniques to satisfy breakpoint conditions that span across multiple UDFs. It incorporates various optimization techniques to reduce the runtime overhead. We conduct experiments with multiple UDF workloads on various datasets and show its high efficiency and scalability.

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

Additional Key Words and Phrases: User-defined functions (UDFs), debugging, big data systems

ACM Reference Format:
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 (SIGMOD), Article 225 (December 2023), 26 pages. https://doi.org/10.1145/3626712

1 INTRODUCTION

Big data systems (e.g., [2, 6, 41]) have gained increasing popularity due to their ability to process large and complex datasets. One of the main features of these systems is their ability to incorporate user-defined functions (UDFs), which allow users to create functions with their customized logic [12, 22, 26, 27, 31]. UDFs extend the usability of these systems, particularly for complex data-processing jobs, where the built-in functions may not be sufficient to handle the data in the way required by the user. Moreover, UDFs allow the integration of existing code and libraries written in other programming languages, such as machine learning libraries in Python or statistical libraries in R, into these systems [29, 40].

Due to their complexity, developing a UDF may require significant effort, especially when it comes to debugging. Even minor errors in the UDF code can result in unexpected outcomes for a data-processing job. For instance, consider a workflow shown in Figure 1, which retrieves real-time tweets through a Twitter API, labels the city name in each tweet, identifies the corresponding timezone based on the city, and saves the output into a file.

Authors' addresses: Yicong Huang, yicongh1@ics.uci.edu, University of California, Irvine, USA; Zuozhi Wang, zuozhiw@uci.edu, University of California, Irvine, USA; Chen Li, chenli@ics.uci.edu, University of California, Irvine, Department of Computer Science, Donald Bren Hall, Irvine, California, USA, 92697-3435.

2836-6573/2023/12-ART225
https://doi.org/10.1145/3626712

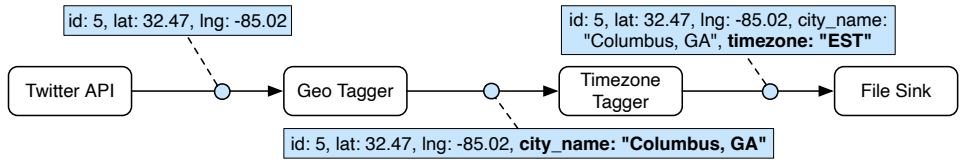

Figure 1. A workflow to tag the geolocation and timezone on a stream of tweets. An example tweet tuple is shown in blue, with the newly added fields highlighted in bold after each operator.

As shown in Figure 2, the GeoTagger operator is implemented as a Python UDF, which infers the city of the tweet based on the tweet's latitude and longitude. Unfortunately, this implementation has several vulnerabilities, such as:

i) Data errors: If the `lat` field is absent from the input tweet, the `r_tree.query()` function may raise exceptions or return empty results.

ii) Code errors: The for-loop gradually increases the search radius to find the closest city but does not terminate after a city is found. Thus the loop may continue and identify another city when a larger radius is provided, leading to an incorrect output.

```
1  def geo_tagger(r_tree, cache, tweet):
2    lat, lng = tweet.lat, tweet.lng
3    city = cache.get((lat, lng))                # fetch (lat, lng) from cache
4    if city is None:                            # no cached value found
5      for radius in range(1, 10):               # search for the closest city
6        city = r_tree.query((lat, lng), radius)
7    cache.set((lat, lng), city)                 # update the cache
8    return tweet + {'city_name': city}
```

Figure 2. A UDF implementation of the GeoTagger involves searching for latitude and longitude and then expanding the search radius until it finds the closest city.

One common method to debug UDFs on a data-processing system is analyzing logs [7, 30]. For instance, to identify the code error that produces the wrong city name, the developer can inspect the `city` object by adding a logging statement after line 6, e.g., `print(city)`, and analyzing the log records to see if there is an incorrect output. However, this method has several known limitations. *i)* When a UDF is processing a large dataset, the generated log entries can be large. *ii)* Logging statements introduce significant overhead to the UDF. *iii)* The added logs may not provide enough runtime details to show the cause of an error. To obtain more specific information, developers may need to rerun the task after adjusting the logging statements to print additional information, such as the variable `radius` in different loops and steps within `r_tree.query()`. The detailed log records increase the analysis complexity and performance overhead.

An alternative method is to test the UDF in a development environment with sample data [14]. It allows developers to use a local language debugger to trace the UDF code interactively. For instance, by setting a breakpoint at line 6, the developer can inspect the `city` object and step through the for-loop to observe the changes. She can also step into the `r_tree.query()` function to inspect the search details. The sample-based method has its own limitations. First, sample data may not fully represent all possible scenarios, such as missing or incorrectly encoded values. Second, problems that occur only on a large amount of data, such as those caused by outliers, may not be producible using the sample data.

In this paper, we study the problem of how to enable an interactive debugging experience for UDFs *within* data-processing systems. Specifically, we aim to discover how to support various runtime debugging operations, such as setting breakpoints on code lines to break the execution of a UDF while it runs in the data system. For instance, we want to set a breakpoint at line 6 of the GeoTagger UDF to inspect the `city` object. Once the execution pauses, we want to navigate through the UDF code and step into function calls such as `r_tree.query()`. If there is a bug, we also want to apply fixes to the UDF code without killing and restarting the job. Additionally, when a tuple is processed by multiple UDFs that have dependencies or correlations between each other, we want to set breakpoints with conditions related to more than one UDF.

When developing a solution, we aim to leverage existing language debuggers (e.g., gdb, jdb, or pdb) to achieve the goal without re-inventing the wheels and benefit from their future improvements. When adopting these debuggers in a data-processing system, we face several challenges. First, in a traditional setting, a language debugger controls the execution of a program. In a data-processing engine, a coordinator manages each operator's lifecycle, including UDFs. Thus it is challenging for the coordinator and the debugger to work together to control the UDF execution. Second, it is well known that debugging can introduce overhead to the execution [5, 15, 37]. This overhead can be even more significant on large datasets. For instance, in one of our experiments, a language debugger introduces a 2 to 5 times overhead to the execution of a UDF. We need solutions to reduce the debugging overhead so that the execution of a data-processing job is efficient. Third, a language debugger typically interacts with only one UDF process. However, in a complex workflow scenario, there can be multiple UDFs concurrently running in distinct processes. This multi-process environment necessitates the development of a comprehensive solution that empowers users to debug multiple UDFs, ensuring efficient troubleshooting and analysis across the entire workflow.

We tackle these challenges and present Udon, a novel debugger designed explicitly for UDFs in data-processing systems. Due to the wide adoption of Python, we focus on this language as an example. Udon integrates existing Python debuggers as black boxes and can work with any standard bdb-based debuggers for Python [3, 23, 25]. Udon enables an on-demand debugging experience by offering the flexibility to attach and detach a language debugger to a UDF at any time during the execution of a workflow. During debugging, the debugger and the coordinator communicate with each other, allowing the debugger to fully control the UDF process while keeping the coordinator informed and in control. To minimize the performance impact of debugging, Udon takes data-related conditions into consideration and reduces the overhead introduced by language debuggers. Additionally, Udon supports coordinated debugging of multiple UDFs by introducing state transfer between multiple debuggers.

Specifically, we make the following contributions.

- We analyze two approaches to integrating existing language debuggers into data-processing engines and demonstrate the advantages of running debuggers under the control of the data engine (Section 3).
- We propose a debugger-aware UDF execution model that ensures the responsiveness of the operator during debugging. We also discuss methods to send debug instructions to a UDF operator not in the debug mode and attach a debugger on the fly (Section 4).
- We analyze two sources of debugging overhead on UDFs and propose two techniques for improving the debugging performance (Section 5).
- We discuss debugging scenarios that involve multiple UDF operators and propose methods for inter-operator debugging. We show active and passive methods to support state transfer by sharing intermediate line states between multiple debugger instances (Section 6).
- We conduct an extensive experimental study to evaluate the performance of Udon with multiple UDF workloads on various datasets to demonstrate its high efficiency and scalability (Section 7).

1.1 Related Work

UDF support in data-processing systems. Many data-processing engines, such as Apache Spark (PySpark [27]) and Apache Flink (PyFlink [26]), enable users to write Python UDFs and integrate them into data-processing pipelines. While certain engines offer limited ability to use external debuggers [10, 11], they lack comprehensive debugging features integrated into the engine itself. In a recent survey [12], a taxonomy of methods for integrating UDFs into data engines is discussed, with the majority of approaches [16, 33] prioritizing the optimization of UDF performance over debugging support. Udon deeply integrates language debuggers into the data-processing engine, providing native debugging awareness and built-in support for debugging operations.
Debuggers in big data systems. BigDebug [13] provides debug primitives for Apache Spark and allows users to insert simulated breakpoints that record all input and output data of operators. This feature helps users pinpoint the locations of errors. TagSniff [8] uses a tag-based mechanism to label and trace the data flow between operators. ML-PipeDebugger [17] offers an overview of the behavior of specific data instances or pairs of data instances as they move through the pipeline. These tools primarily focus on debugging input and output data, rather than the intermediate states of UDFs. They treat operators as black boxes and do not support line-by-line debugging inside a UDF. Port [20] supports out-of-place online debugging, which involves transferring the debugging session to an external process, allowing developers to debug in an isolated environment. Pedro et al. [14] export UDF code and its input data from a database to the client side, enabling developers to execute UDF code with a local debugger. These methods lose efficiency when handling large dataset transfers and are not applicable if the bug only occurs in the original environment but not locally. Udon focus in debugging UDF operators within their original data-processing environment.
Online debuggers for distributed systems. For example, GoTcha [1] focuses on observing state changes in distributed systems to support interactive debugging. MaceODB [9] provides runtime properties and checks for violations during an execution. Once a violation is detected, the tool provides the programmer with information to determine the cause. The algorithm proposed in [21] extends the Chandy-Lamport global-state-recording approach to detect distributed breakpoints and maintain consistent states, addressing issues with partially ordered events and communication in distributed debugging. These solutions do not support user-defined functions and do not offer line-by-line debugging primitives.
Other debuggers for distributed systems. Oddity [39] provides developers with fine-grained control over messages exchanged between distributed nodes. It also supports time travel, enabling developers to explore multiple branching executions of a system. Chukwa [30] monitors Hadoop clusters at runtime and stores log data for analyzing runtime failures. SALSA [34] analyzes log records produced by a distributed system and constructs a state machine to help developers debug the execution. While these tools allow developers to monitor, inspect, or perform post-mortem analysis of failures, they do not support real-time line-by-line debugging in distributed systems.

2 PROBLEM SETTING

2.1 Data Workflows and User-Defined Functions

A data workflow is a directed acyclic graph (DAG) that consists of operators as vertices and edges for data transfer. The workflow is executed in a data-processing engine to compute its results. A user-defined function (UDF) is a piece of code provided by a user, and processes data tuple by tuple. Typically, a UDF operator can be written in a language different from the engine's language. As a consequence, the operator's execution may occur in a different runtime environment than that of the engine. In this paper, we consider the common case where the data-processing engine is written in Java/Scala, and it uses a Java virtual machine (JVM) as its runtime environment. We

assume UDF operators are written in Python due to their wide adoption in the field of data science, data analytics, and machine learning. Python UDF execution necessitates a Python virtual machine (PVM) as the runtime environment.

UDF execution architecture. The execution of the workflow is managed by a component called *coordinator*. It sends control instructions through control channels to each operator for purposes such as initialization, termination, or cancellation of the operator. Each operator has a receiver and a sender responsible for message passing, which transmit control instructions between the coordinator and operators, as well as send data tuples among operators. Each operator also manages a computation function that continuously processes the received data tuples. For UDF operators, the engine has a proxy that forwards the input tuples to the operator's receiver in the PVM. The operator's sender returns its output tuples to the proxy, which then forwards them to the downstream operator. Figure 3 shows a Scala engine such as PyFlink and PySpark that executes the aforementioned workflow with the described architecture. The Twitter API operator, the Timezone-Tagger operator, and the File Sink operator, as well as the coordinator, are running on the same JVM. The GeoTagger operator is a Python UDF operator and it runs in a PVM.

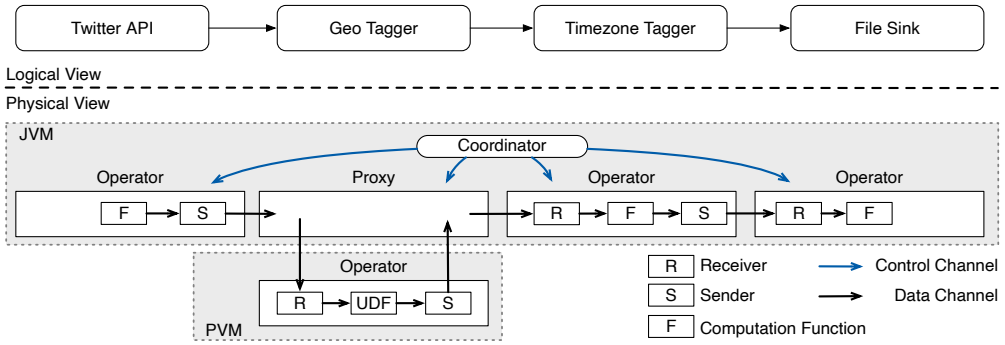

Figure 3. Execution of a Python UDF in a Scala engine.

2.2 Runtime Debugging Primitives of UDF

We want to support debugging of a UDF with common primitives of language debuggers, including the following operations.

Execution Control. *i) Breakpoint:* A user-specified location in the code where the debugger will pause execution and allow the user to inspect variables, call stack, and other runtime states. *ii) Step:* A command that advances the execution by a single step. This can be useful for debugging loops or complex code. *iii) Continue:* A command that resumes the execution of the program from the current breakpoint.

Inspection. *i) Watch:* A user-specified variable continuously monitored by the debugger, displaying its value when it changes. *ii) Evaluate:* A command that allows the user to evaluate an expression and view its value. *iii) Print:* A command that prints a variable as a string.

Fault Handling. *i) Fault culprit:* In extreme scenarios, the computation could have a fault on a specific input tuple, such as an unhandled runtime exception. The primitive is used to identify the root cause of faults in the debuggee UDF. *ii) Updating computation function:* This primitive allows users to update the code of a UDF during runtime without having to restart the entire system. *iii) Updating intermediate states:* This primitive allows users to modify the intermediate states of a UDF during runtime.

2.3 Existing Runtime Python Debuggers

We consider runtime language debuggers such as pdb [23] and PyDev.Debugger [25]. Figure 4 demonstrates how such debuggers operate. The user issues instructions to the debugger through a front-end, which could be a command-line interface or a graphical interface. These instructions may involve investigating intermediate states, stepping through the code, and dynamically adding or removing breakpoints. Once a breakpoint is received, the debugger records it in a breakpoint-lookup table for later references.

```
4  if city is None:
5      for radius in range(1, 10):
6          city = r_tree.query((lat, lng), radius)
7  cache.set((lat, lng), city)
```

breakpoint-lookup table	
file_name	**line_no**
geo_tagger.py	6
…	…

Figure 4. Using a Python debugger to debug the GeoTagger. User's breakpoints are recorded in an in-memory table for later references and checks.

During debugging, the debugger listens to trace events (e.g., `PyTrace_Line`) emitted by the Python interpreter before interpreting a statement line. The debugger examines each code line and looks up the breakpoint table to check if a breakpoint is hit. If so, the debugger suspends the execution of the current thread and prompts the user for the next instruction. In the absence of a breakpoint, the debugger proceeds, permitting the following line of code to execute. Throughout the remainder of this paper, we will employ pdb as an illustrative runtime debugger, with our findings generally extending to other runtime debuggers.

3 INTEGRATING A DEBUGGER INTO THE ENGINE

In this section we discuss two approaches for integrating debuggers into the data-processing engine to control a UDF's execution.

3.1 Approach 1: The Debugger Runs Separately from the Engine

Figure 5 shows the first approach, in which a debugger runs separately from the engine. Users interact directly with the debugger through a front-end. With pre-configured breakpoints, they await debug events (e.g., breakpoint hits) and then send instructions, such as variable evaluation, and receive responses from the debugger. Users can also instruct the debugger to continue execution until the next event. This direct connection bypasses the data-processing engine, which remains unaware of the debugger's presence. This approach is widely adopted in systems that support Python UDFs, such as PyFlink [10] and PySpark [11]. This approach comes with several disadvantages.

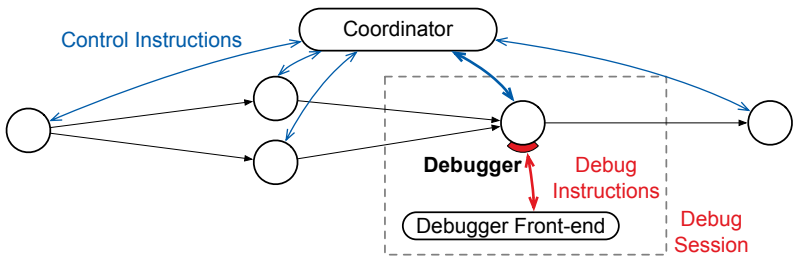

Figure 5. Approach 1 - The debugger runs separately from the engine. The debug session is limited to the debugee operator.

Unresponsiveness caused by two control sources. When a breakpoint is hit, the debugger pauses the execution and awaits the user's next instruction. During this potentially extended waiting period, the operator becomes unresponsive to any instructions from the engine, resulting in several side effects. For instance, if the engine sends a heartbeat message to the operator to check its aliveness, since the operator is incapable of responding, the engine may erroneously conclude that the operator has encountered a failure. Consequently, the engine might activate a fault-recovery mechanism to terminate and restart the unresponsive operator. This unresponsiveness arises from the fact that the same operator is under the control of two separate sources: the coordinator and the debugger, with neither being aware of the other's actions.

Uncontrolled workflow suspension. When an operator hits a breakpoint, a debugger suspends its execution. Although it doesn't directly suspend other operators in the workflow, the halted operator can trigger a ripple effect, ultimately leading to the suspension of the entire workflow. In particular, downstream operators will pause their execution after consuming all in-flight tuples because the suspended operator no longer generates new tuples. Upstream operators will continue sending tuples, but as the debuggee operator's input buffer fills up, these operators must also suspend to prevent buffer overflow using a backpressure mechanism. The backpressure effect will gradually propagate to all upstream operators. It's important to note that the workflow does not consume any CPU resources when suspended and it can yield the CPU resources to other workflows. This is because the operators are blocked based on specific conditions such as breakpoints or full buffers, rather than engaging in busy waiting. However, this ad-hoc suspension process lacks systematic management and control. Consequently, it may take a considerable amount of time for the entire workflow to suspend, potentially leading to the consumption of a significant amount of memory due to data buffering.

Lack of synchronization between multiple debuggers. When a workflow incorporates multiple UDF operators or several instances of the same operator, the engine may initiate a PVM for each UDF operator, particularly when executing on a cluster. Each PVM requires a debugger and a front-end, and the absence of global synchronization between these debuggers can lead to issues. For instance, suppose the user intends to debug two parallel instances of the same operator, denoted as A and B. The user sets a breakpoint on the GeoTagger operator at line 6 with the condition `tweet_id == 1` for both A and B. Instance A hits the breakpoint and pauses at line 6 when processing a tweet with `id == 1`. Now, the user identifies a bug and updates the GeoTagger's code, which should be reflected in both instances. Unfortunately, while A is paused at the breakpoint and can accept the code-change instruction, B doesn't receive this instruction because it never reaches the breakpoint. In general, the multi-operator setting complicates the coordination of debuggers with each other.

3.2 Approach 2: The Engine Controls the Debugger

We propose a new approach as illustrated in Figure 6, in which the debugger operates under the engine's control.

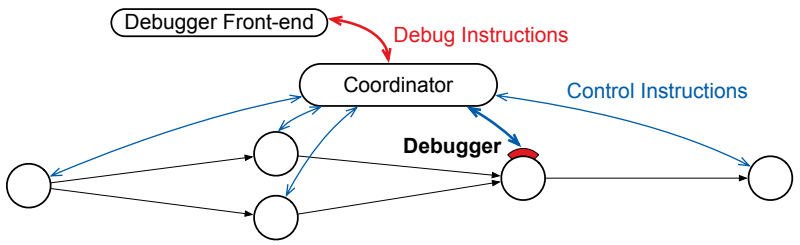

Figure 6. Approach 2 - The debugger is controlled by the engine. The debug session is visible to the coordinator.

The engine uses a single debugger front-end for all its debuggee PVMs. It allows the front-end user to send debug instructions to the coordinator at any time, which subsequently relays the instructions to the target debugger. Once the instruction is received, the target debugger processes it and sends a response back to the coordinator, which forwards it to the front-end. Each debugger can also emit debugging events (e.g., breakpoint hits) to the front-end through the coordinator. In comparison to the previous approach, this approach offers the following advantages.

A single source of control. As the coordinator is the sole control source and is aware of all the debuggers, operators are never placed in conflicting states. Consequently, the problematic unresponsiveness situation detailed in Section 3.1 can be effectively prevented.

Coordinated workflow suspension. After an operator hits a breakpoint, the coordinator suspends all the operators in the workflow by broadcasting a pause instruction. Each operator, except the debuggee operator, gracefully suspends itself. Additionally, each operator can choose the optimal suspension point to conserve resources during the suspension and ease the resumption process. For example, an operator could suspend itself after completing the processing of its current tuple, which may release all intermediate states for that tuple.

Supporting synchronization between multiple debuggers. The coordinator maintains a global view of all the debug instructions to the debuggee PVMs. In the example where the user wishes to update the code for the GeoTagger operator with two instances A and B, upon receiving a code-change instruction, the coordinator can broadcast the instruction to both A and B, effectively synchronizing these debuggers.

In the rest of the paper we will focus on approach 2 due to its benefits.

4 SENDING DEBUG INSTRUCTIONS TO A UDF

In this section, we delve into the process of sending a debug instruction to a UDF operator and how it's processed during UDF execution. We use the common debug instruction of setting a breakpoint to explore different options and discuss their pros and cons.

One approach is to add the debug instruction as a line of code within the UDF and execute the code from the beginning. For example, as illustrated in Figure 7, a user modifies the code by adding a `breakpoint()` statement before the line she wishes to investigate, then restarts the execution. The `breakpoint()` statement acts as a hook: when encountered during execution, it invokes pdb. This suspends the execution on the subsequent line and prompts the debugger front-end for the next instruction.

```
5  for radius in range(1, 10):
6      breakpoint()
7      city = r_tree.query((lat, lng), radius)
```

Figure 7. Adding a breakpoint() statement before the target line to investigate. The line number of the target line (e.g., line 6) is subsequently incremented (e.g., to line 7).

A major limitation of this method is that modifying the source code requires killing and restarting the UDF execution and the workflow. Next we discuss methods for dynamically sending breakpoints to a debugger without restarting the execution.

4.1 A Novel UDF Execution Model

We propose a two-thread execution model for a UDF operator to enable dynamic reception of debug instructions. As shown in Figure 8, in this model, each UDF operator consists of two threads: a control-processing (CP) thread and a data-processing (DP) thread. These threads communicate

through shared variables. The CP thread is responsible for receiving control instructions from the coordinator. In line with the architecture detailed in Section 3.2, the CP thread receives all control instructions, including standard 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 is managed by pdb.

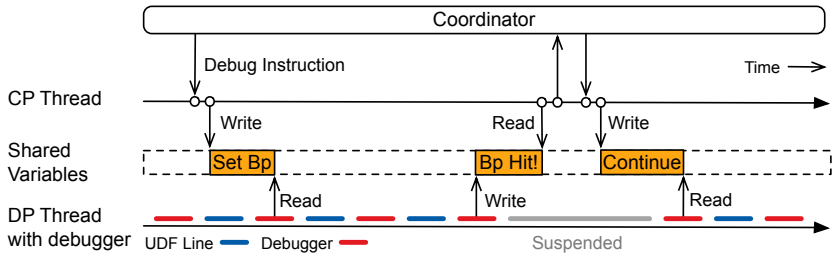

Figure 8. The proposed two-thread UDF execution model with a debugger. The control-processing (CP) thread and the data-processing (DP) thread exchange debug instructions and events through shared variables.

Handling debug instructions. Figure 8 also illustrates the lifecycle of this model handling debug instructions with an example. First, when the CP thread receives a `set breakpoint` command, it updates a shared variable. During the execution of the DP thread, pdb processes trace events between every two lines of code to check the shared variable and register any breakpoints. When a breakpoint is hit, pdb 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. During this suspension, the DP thread remains blocked at the exact code line where the breakpoint was hit, incurring no additional CPU consumption. Simultaneously, the CP thread can still receive and process other control instructions from the coordinator, such as heartbeat and statistics checks. When the user sends a `continue` instruction, the CP thread similarly notifies pdb, which resumes the DP thread to continue the execution of the operator's data-processing logic.

Handling faults and exceptions. Similar to a breakpoint-hit event, when an exception is raised from the UDF code during runtime, pdb captures and reports it as a debug event. This event is then forwarded to the coordinator and eventually presented to the debugger front-end. At this point, the user's UDF runtime has stopped, but the DP thread remains active with pdb attached. The user can then engage in post-mortem debugging [23] to inspect the exception. During post-mortem debugging, the user has the flexibility to inspect the operator's internal states, the data being processed, and other system-related information. After inspection, the user can make necessary updates to the faulty UDF code, state, or data. The new UDF code will be recompiled and used to replace the old, flawed code in place, with the internal states of the operator retained and accessible to the new UDF. Following these adjustments, the user can resume execution with the fixed code or data by restarting the UDF within the DP thread.

A key advantage of this execution model is that it enables the debugger to control the data-processing execution in the DP thread while allowing the CP thread to run independently. This design aligns with the requirements described in Section 3.2.

4.2 Efficient Passing of Debug Instructions between Threads

The design outlined in Section 4.1 mandates that the DP thread continuously operates in debug mode and uses trace events to monitor incoming instructions. This method introduces significant

overhead, even if no instruction is ever received. A natural question arises: can we allow the DP thread to run in its regular mode when there are no instructions, and only switch to debug mode when an instruction arrives? Next, we explore two approaches to implement this switching and reduce the associated overhead.

4.2.1 Signal-based notification. This approach uses the `SIGINT` signal available in Unix-based systems. As illustrated in Figure 9, the DP thread initially executes the UDF code without debuggers. The CP thread first writes the instruction in the shared variable and then sends a `SIGINT` signal to notify the DP thread. When the DP thread receives the signal, its handler invokes pdb to take control of the execution, and the following UDF code is wrapped with the debugger's code to execute. The debugger then processes the stored instruction immediately.

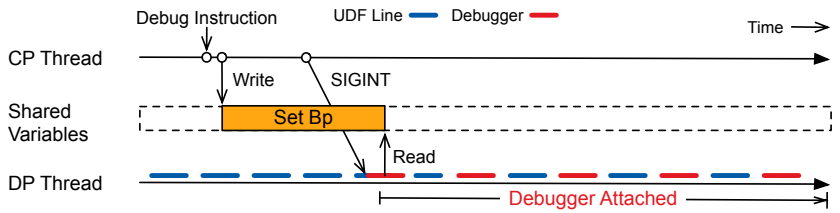

Figure 9. The CP thread sends a signal to notify the DP thread of a debug instruction, which could take some time to arrive. The DP thread attaches the debugger to process the instruction after receiving the signal. The signal will be handled in the next bytecode, which may be in the middle of a UDF code line.

The signal-based approach has several advantages. First, the DP thread does not need to execute additional code to receive the signal, thus it has a low overhead when no instructions are sent. Second, it provides a fast response time. Since signal handlers in Python are executed before the next bytecode, they offer a fine granularity for handling debug instructions. This approach has the following limitation. A signal to a Python process [28] can only be delivered to the main thread (i.e., the first thread of a Python process) of PVM [38]. Other threads spawned by the main thread cannot receive and process the signal. To use this approach, the DP thread needs to be the main and the only thread of PVM. This limitation reduces the applicability of the approach. For instance, many systems employ multiple threads to process data in parallel [42], making it impossible to use signals to notify debug instructions for a specific DP thread.

4.2.2 Explicit checking. Another approach is illustrated in Figure 10. Between the processing of two tuples, the UDF code finishes and lets the DP thread execute system instructions (e.g., outputting results and fetching the next tuple). Here, an explicit check for debug instructions can be added.

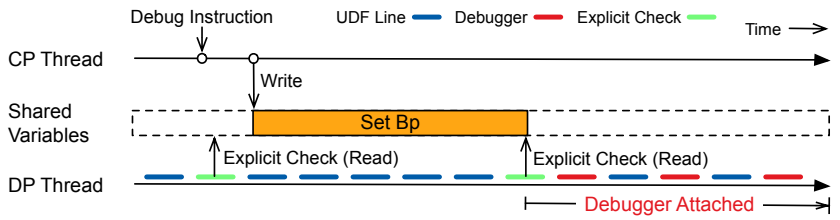

Figure 10. The DP thread explicitly checks for debug instructions. Once an instruction is available, the DP thread attaches a debugger to process it.

Users can also explicitly instruct the DP thread to check for debug instructions at a desired position within the execution of the UDF code. For example, the user can add the following statement in the UDF at the desired position to check for instructions:

```
if instruction != None: process(instruction)
```

In Python, users can utilize the `yield` statements to write the UDF code as an iterator with multiple mini-steps. This way enables the DP thread to perform instruction checks between the execution of two mini-steps within the iterator. It allows the DP thread to inspect debug instructions at a granularity specified by the user, providing users with full control over the debugging process.

This explicit checking approach offers more flexibility than the previous one because it doesn't impose any restrictions on the threading model. However, it has two drawbacks. First, the interval between two consecutive explicit checks can be lengthy. If a debug instruction arrives between two explicit checks, it might need to wait for an extended period before being handled at the next explicit check. Second, to reduce the interval, the user may add multiple explicit checks to the UDF code, which results in additional runtime overhead. In contrast, the signal-based approach does not incur any checking overhead.

5 IMPROVING UDF DEBUGGING PERFORMANCE

Debugging's impact on UDF operator performance is crucial, particularly for large data processing. In this section, we analyze debugging overhead sources and develop optimization techniques.

5.1 Two Sources of Debugging Overhead

Debuggers of a UDF introduce overhead as they add many extra instructions to examine a UDF's internal states and collect information about its execution. We use breakpoints as an example to understand the overhead. After the breakpoint is set, a debugger stores its corresponding file name, line number, and conditions into a breakpoint-lookup table. During the subsequent execution, the debugger looks for potential breakpoint hits on every statement line, which includes two types of instructions: breakpoint checks and predicate evaluations. Consider the GeoTagger operator with a breakpoint set on line 6, we show two types of instructions in Figure 11.

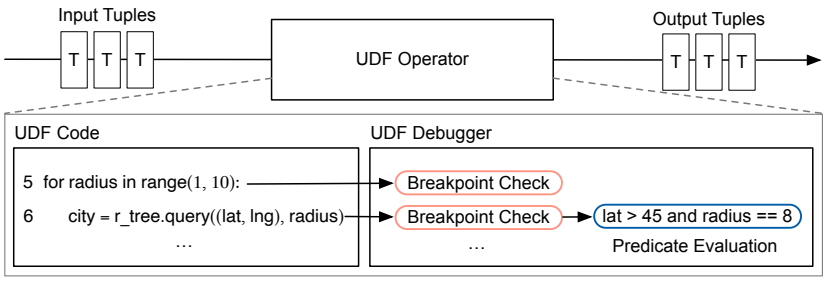

Figure 11. UDF's overhead of handling breakpoints while processing tuples, including breakpoint checks (marked in red) and predicate evaluations (marked in blue).

Breakpoint checks. Before each statement line, the debugger performs a breakpoint check to retrieve any breakpoints set on the line. Specifically, it listens to trace events (e.g., `PyTrace_Line`) emitted by the Python interpreter before interpreting each statement line. In each check, the debugger queries the breakpoint-lookup table using the file name and the line number. Note that the debugger needs to perform breakpoint checks on every statement, regardless of whether a breakpoint is set on the line. Moreover, the checks are repeated for lines interpreted multiple times, such as statements within a loop.

Predicate evaluations. A breakpoint might contain one or more predicates whose evaluation incurs additional overhead. A predicate can be a flag indicating whether a breakpoint is disabled or a user-defined condition from a conditional breakpoint with a certain evaluation complexity. For instance, one may want to evaluate `r_tree.contains((lat, lng), radius)` in the conditional breakpoint, as such a tree traversal can be expensive. Similar to breakpoint checks, the debugger repeatedly evaluates predicates because the variables used in the predicates can change.

Both breakpoint checks and predicate evaluations can substantially increase the overhead of processing data using the UDF. As the data volume increases, the overhead can grow rapidly. Next, we develop two optimizations to reduce the overhead.

5.2 Optimization 1: Reducing Breakpoint Checks by Hot-swapping UDF Code

This optimization reduces breakpoint checks by "hot-swapping" the UDF code with a modified code that adds breakpoint hooks as new statement lines. As Python is a dynamic language, the native language debugger cannot swap the code while the interpreter is using it. In a data-processing engine, however, we can take advantage of the fact that the interpreter does not use the UDF code between processing two tuples. This location is natural for performing hot-swaps without disrupting the ongoing workflow execution.

As shown in Figure 12, since breakpoints are explicitly anchored at the designated lines to break, there's no need to check for breakpoints on every line. The statement invokes the debugger to break only when a predicate evaluates to true and skips breakpoint checks on irrelevant lines of code. Therefore, this optimization reduces the number of unnecessary breakpoint checks and improves the efficiency and effectiveness of debugging.

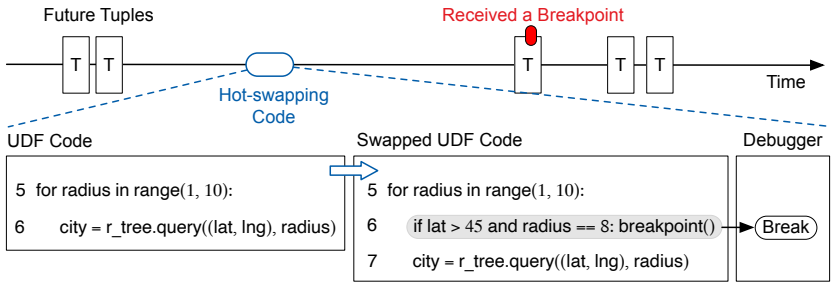

Figure 12. Optimization 1: When a breakpoint is received, the operator hot-swaps the UDF code to incorporate its predicates with breakpoint() hooks after processing the current tuple. Subsequently, these predicates are evaluated for upcoming tuples, and the debugger is triggered if they evaluate to true for a tuple.

Applying the optimization. When a breakpoint is set, the debugger generates and compiles a new version of the UDF code that includes `breakpoint()` hooks, which invoke the debugger to break at their subsequent lines. Before executing the next tuple, the debugger hot-swaps the original code with the new code and transfers all the internal states of the operator. Then, the debugger is configured to ignore trace events and instead listen to `breakpoint()` hooks in the code. This setup can process tuples continuously until a tuple requiring a different optimization arrives, at which point the debugger can easily disable this optimization by hot-swapping back the original code and transferring the new internal states back.

Cost of the optimization. This method pays a one-time cost for the compilation of the new code and configuration of the debugger. No additional overhead is introduced for each new tuple processed with this optimization. As all the predicates are written in the code as statement lines, they will still be evaluated and incur overhead.

5.3 Optimization 2: Improving Evaluations by Pulling up Predicates

We develop another optimization to reduce the overhead of conditional breakpoints by distinguishing two types of predicates: (1) predicates set on intermediate variables (e.g., the `radius` in the GeoTagger operator) and (2) predicates set on incoming tuples. This technique pulls up the second type of predicates, evaluates them directly on the incoming tuples instead of the target statements, and stores the evaluated results for future reference.

As illustrated in Figure 13, the example predicate `lat > 45` is using the data field "lat", thus it is pulled up to be evaluated on the tuple before the tuple is processed. This optimization reduces overhead in two ways. First, if the pulled-up predicate on a tuple short-circuits the other predicates, all breakpoint checks when processing the tuple can be skipped, and the tuple can be processed by the UDF without a debugger. Second, this optimization can reduce the number of repeated evaluations for breakpoints set in a loop or a repeatedly invoked function, especially for predicates that have a high evaluation cost.

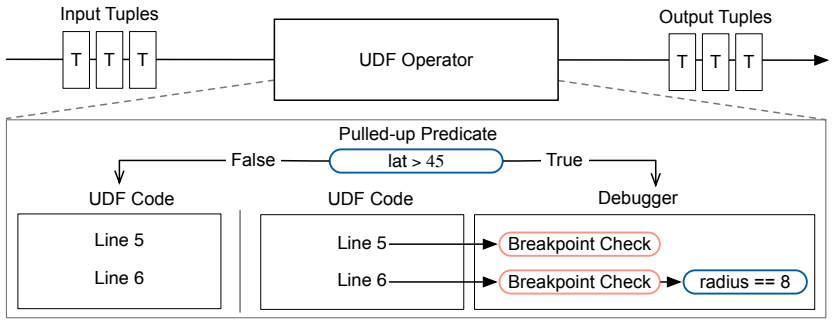

Figure 13. Optimization 2: Pulling up a predicate of a breakpoint to be evaluated before processing an input tuple with the UDF code. Depending on the evaluation result, some predicate evaluations and breakpoint checks can be short-circuited, and the tuple may be executed with or without a debugger.

This optimization is particularly viable in a data-processing engine as we can distinguish between intermediate variables and input tuples. Predicates set on input tuples can be evaluated early because the content of the input tuple is immutable, while predicates set on intermediate variables cannot be evaluated early because their values might change during the UDF execution. In the example, the predicate `radius == 8` must be evaluated on statement line 6 because the value of the intermediate variable `radius` can keep changing after each iteration. As Python is a dynamic language, identifying whether a variable is immutable is impossible, making it impractical to pull up predicates using intermediate variables.

Additionally, we can leverage the data properties and distributions from the upstream operators of a UDF operator to further reduce evaluations on the pulled-up predicates. For instance, consider adding a Filter operator with a selection predicate `tuple.lat < 30` before the GeoTagger operator. This ensures all input tuples for the GeoTagger operator have a latitude of less than 30. Consequently, we know that none of the input tuples will satisfy the pulled-up predicate `lat > 45`, allowing us to skip its evaluation for all the input tuples of this operator. Similarly, consider a pulled-up predicate `id == 123`, and the input data is known to be unique on the id field. In this case, only one tuple could satisfy this predicate. Once the debugger encounters the tuple with an id equal to 123, the predicate evaluations on all future tuples can be skipped. Many other data properties can be used as helper predicates as well, including but not limited to sorted order, partition key, etc.

Applying the optimization. Upon receiving a breakpoint, the operator identifies predicates on incoming tuples and can be pulled up with respect to itself. To process a tuple with the pulled-up predicates, it first evaluates the predicates with the input tuple and stores the results in a predicate evaluation table. If a predicate short-circuits a breakpoint, the operator temporarily disables it by removing it from the breakpoint-lookup table in the debugger. Otherwise, it updates the predicates of the breakpoint to avoid redundant evaluations by performing a lookup from the predicate-evaluation table for the pulled-up predicate during line-by-line checks and evaluations. Once the tuple has been processed, the disabled breakpoints are added to the breakpoint-lookup table and re-enabled by the optimization. This process repeats for each incoming tuple.
Cost of the optimization. Compared to optimization 1, which needs a one-time compilation cost, this optimization technique requires making runtime decisions for each tuple based on its data and predicates. Specifically, before processing a tuple, it spends time to evaluate pulled-up predicates and modify or remove breakpoints in the debugger according to the evaluation results. While the cost per tuple is negligible, the time increases linearly with respect to the number of tuples.

6 DEBUGGING MULTIPLE UDF OPERATORS

So far, we have discussed how to debug a single UDF operator. In this section, we study how to debug multiple UDF operators. We will first use an example to motivate the problem, then develop several methods to solve it.

6.1 The Need of Inter-Operator Debugging

Suppose the code error in Figure 2 is fixed by using a while-loop to replace the if-statement and the for-loop, as shown in Figure 14a. In addition, assume that the TimezoneTagger operator in Figure 1 is also implemented as a UDF, with the code presented in Figure 14b. It first searches for the timezone using `timezone_api` with each input tweet's coordinates. If it fails, e.g., due to a "location not found" error, it re-attempts the search with the tweet's city name tagged from GeoTagger.

```
1  def geo_tagger(r_tree, cache, tweet):
2    lat, lng = tweet.lat, tweet.lng
3    city = cache.get((lat, lng))
4    radius = 0
5    while city is None: # stop when found
6      radius += 1 # gradually increase range
7      city = r_tree.query((lat, lng), radius)
8    cache.set((lat, lng), city)
9    return tweet + {'city_name': city}
```

(a) The updated GeoTagger UDF (replacing line 4 to line 6) uses a while-loop to stop after finding the city.

```
1  def timezone_tagger(timezone_api, tweet):
2    try: # search with exact geolocation
3      lat, lng = tweet.lat, tweet.lng
4      timezone = timezone_api.get(lat, lng)
5    except: # search with inferred city
6      timezone = timezone_api.get(tweet.city)
7    return tweet + {'timezone': timezone}
```

(b) The TimezoneTagger UDF searches for the timezone of tweets using coordinates and city names.

Figure 14. The updated GeoTagger UDF and the TimezoneTagger UDF implementations.

The TimezoneTagger implementation has an issue: utilizing a city name to search for the timezone can yield inaccuracies, as certain cities encompass multiple timezones. For instance, a tweet originating from the western region of Columbus, GA, in the CST timezone, could be mistakenly classified because the central point of the inferred city, Columbus, employs the EST timezone. To inspect this issue, the user aims to pinpoint these problematic tweets. In a simplified setting, these problematic tweets must meet the following two conditions:

- `C1(radius > 10)`, line 6, GeoTagger (W_{GT})
- `C2(timezone == "EST")`, line 7, TimezoneTagger (W_{TT})

When such a tweet is found, the execution should pause at line 7 of W_{TT} for further inspection. A *line state* is the value of a variable at a certain line of UDF. These two conditions require accessing the line states of the two operators: `radius` of W_{GT} at line 6 and `timezone` of W_{TT} at line 7. This access is challenging because the two line states do not exist simultaneously. In fact, for the same tuple, `timezone` is available later than `radius` because W_{TT} depends on W_{GT}.

6.2 Baseline Method: Tracing a Tuple Manually

The user can use debug primitives from earlier sections to trace tuples and locate problematic tweets. First, the user can set a breakpoint on line 6 of W_{GT} with condition C1 to catch the next tweet that satisfies the condition:

$$Bp_1\text{: C1, at line 6 of } W_{GT}.$$

Then she traces this tweet downstream in W_{TT} to inspect and check if it meets condition C2. To accomplish this task, she utilizes the identifier of the tweet to set a second breakpoint on line 7 of W_{TT} with a new condition C3, enabling her to trace the tuple from W_{GT} to W_{TT} along the DAG:

$$Bp_2\text{: } \texttt{C3(tweet.id == 123)}\text{, at line 7 of } W_{TT}.$$

Now she has two breakpoints, and the workflow is paused. She disables Bp_1 and resumes all operators so that the tweet can pass W_{GT} and move to W_{TT}. When it reaches W_{TT} at line 7, Bp_2 is hit. She evaluates C2 to determine if the tweet satisfies it. If so, she finds a problematic tweet. The user disables both breakpoints and repeats the steps to trace and capture additional tweets that meet the conditions. In general, the steps of this method are tedious and painstaking.

6.3 Advanced Method: Coordinated State Transfer

We propose a method in which the coordinator handles a breakpoint with a global condition by passing line states between different UDFs. In the example, a breakpoint with the following global condition can be sent to the coordinator:

$$\texttt{C1((}W_{GT}\texttt{, line 6, radius) > 10) AND C2((}W_{TT}\texttt{, line 7, timezone) == “EST”).}$$

As the line states are located inside each operator, to handle the breakpoint, the coordinator work with debuggers to transfer states between operators. When a problematic tweet is found, the coordinator notifies the user and pauses all the operators. This method addresses the limitation of the baseline method by only requiring the user to send a single breakpoint with the global condition. The user is prompted only for tuples that satisfy the global condition and will not be exposed to internal debug commands exchanged between debuggers, minimizing the chances of making mistakes. Next, we present two approaches for state transfer.

6.3.1 Active approach: Transfer states through data tuples. In this approach, upstream operators annotate data tuples with line states and send the tuples to downstream operators. As shown in Figure 15, after receiving the breakpoint from the user, the coordinator analyzes the condition and identifies the partial conditions and their corresponding operators. To initiate a state transfer, the coordinator sends an EmbedState command with condition C1 to W_{GT} and a Breakpoint command with condition C2 to W_{TT}.

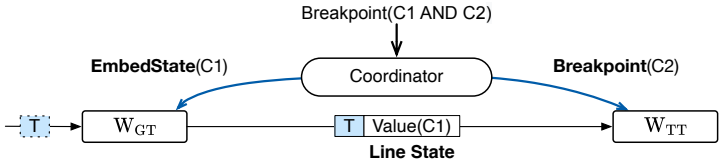

Figure 15. W_{GT}'s debugger actively sends line states to W_{TT} by annotating them on data tuples.

After receiving EmbedState command, W_{GT}'s debugger evaluates condition C1 whenever it hits line 6. If a tuple satisfies C1, instead of pausing execution and notifying the user, the debugger adds the line state `((radius > 10) == True)` to the tuple as new metadata. This metadata is part of the transferred tuple on the DAG and is accessible to downstream operators. After receiving the Breakpoint command, W_{TT}'s debugger evaluates C2 every time it hits line 7 for a tuple. It also checks for the line state of the tuple passed from W_{GT}. If both conditions are satisfied, the debugger pauses the execution and notifies the user of the problematic tuple.

6.3.2 Passive approach: Upstream operators save states for downstream operators to request. In this approach, upstream operators keep line states internally, which can be requested by downstream operators later. As shown in Figure 16, to process the breakpoint with the global condition, the coordinator sends a StoreState command with condition C1 to W_{GT}, and a Breakpoint command with condition C2 to W_{TT}.

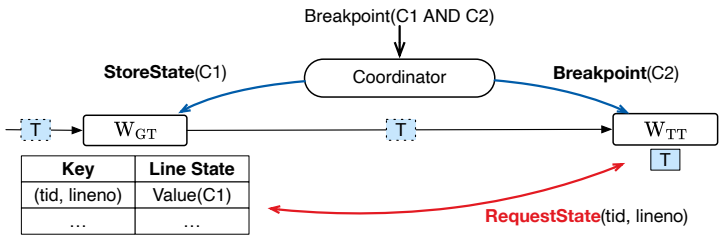

Figure 16. W_{GT}'s debugger stores its line states, which are requested by W_{TT}'s debugger whenever needed.

After receiving the `StoreState` command, W_{GT} debugger evaluates C1 every time it hits line 6. If C1 is true, it stores the line state `((radius > 10) == True)` internally with a reference key. Downstream operators can use the key to retrieve the corresponding line state. The tuple is then processed without interruption in W_{GT} and output to downstream operators with the reference key. After receiving the `Breakpoint` command, the W_{TT} debugger evaluates C2 every time it hits line 7 for a tuple. If C2 is true for a tuple, W_{TT} debugger sends a RequestState command with the reference key to W_{GT} to retrieve the stored line state. If the line state is also true, indicating that the tweet satisfies both conditions, the W_{TT} debugger notifies the user of the problematic tuple.

6.4 Analysis of Two State-Transfer Approaches

We compare the two state-transfer approaches in terms of communication requirements, costs, and delays. We also discuss the suitability of these approaches based on the characteristics of the UDFs.
Communication requirements: The two approaches have different communication requirements for operators. While both approaches require a forward channel for upstream operators to send downstream operators' line states, the passive one additionally requires a backward channel for downstream operators to send state requests to upstream operators. Forward channels (e.g., data

channels) are commonly implemented in modern data-processing systems, whereas backward channels are rarely integrated.

Communication cost: The communication cost of the two approaches depends on *i)* the selectivity of partial conditions, which determines the number of required transfers; *ii)* and the size of each transferred state, which influences the network cost of a transfer. When a partial condition is highly selective, the passive approach is more efficient, as it only needs to retrieve the few tuples that satisfy the condition, while the active approach must transfer the state on all tuples. Conversely, when a partial condition lacks selectivity, the active approach becomes more advantageous, as it does not pause the execution of any tuple. The passive approach is preferable for large state sizes, as it only requires the necessary transfers; for small sizes, the state-transfer cost may be negligible.

Communication delay: The two approaches have different communication delays and thus have different impacts on latency. The active approach has a low latency because it does not pause the execution of upstream or downstream operators. The passive approach requires a downstream operator to send a request to an upstream operator and await a response. As a result, the downstream operator is effectively paused until the upstream operator's response, which can result in a significant delay.

Applicability to UDFs with varied output-tuple cardinalities: The applicability of the two approaches depends on the output-tuple cardinality of the upstream UDF operator. In the running example, the W_{GT} operator executes a UDF that generates one output tuple for each input tuple. In general, if a UDF operator produces one or multiple output tuples per input tuple, both approaches can work in the same way as the example. When a UDF operator produces no output tuple for an input tuple (e.g., a filter or an aggregate operator), the two approaches have different behaviors. The active approach relies on data tuples to transmit annotated line states to downstream operators, thus it can be extended by using a dummy tuple to carry the line state. On the other hand, the passive approach relies on identifiers to reference the state when making state requests. In this case, since the downstream operator receives no tuples, it lacks the identifier needed by the request. As a consequence, the passive approach is ineffective in this scenario.

7 EXPERIMENTS

In this section, we present the results of an extensive experimental evaluation of Udon.

7.1 Setting

Datasets. We used the following datasets in the experiments.

- *Twitter dataset.* It contained about 262,000 tweets obtained from the Twitter platform. The dataset included text content, user, location, and timestamp of each tweet.
- *TPC-H dataset [36].* It consisted of 8 tables about customers, orders, suppliers, line items, and other related data.
- *COCO image dataset [19].* It was a dataset designed for object detection, segmentation, and captioning tasks in computer vision. It included more than 330,000 images with about 2.5 million objects labeled across 80 categories.

Workflows and User-Defined Functions (UDFs). We constructed workflows with Python UDFs shown in Figure 17 to evaluate Udon. Workflows W_1, W_2, and W_5 were taken from real-life projects conducting tweet analysis using NLTK [4] and spaCy [32] libraries. Workflow W_3 was constructed based on TPC-H [36] query 6. Workflow W_4 was constructed to do basic image processing with the Pillow [24] library. Operators including Source, Projection, Join, Filter, and Sink were implemented in Scala/Java, and other operators were implemented as Python UDFs. We selected five representative UDFs (marked as UDF_1 to UDF_5) for a range of common data-wrangling tasks with different complexities.

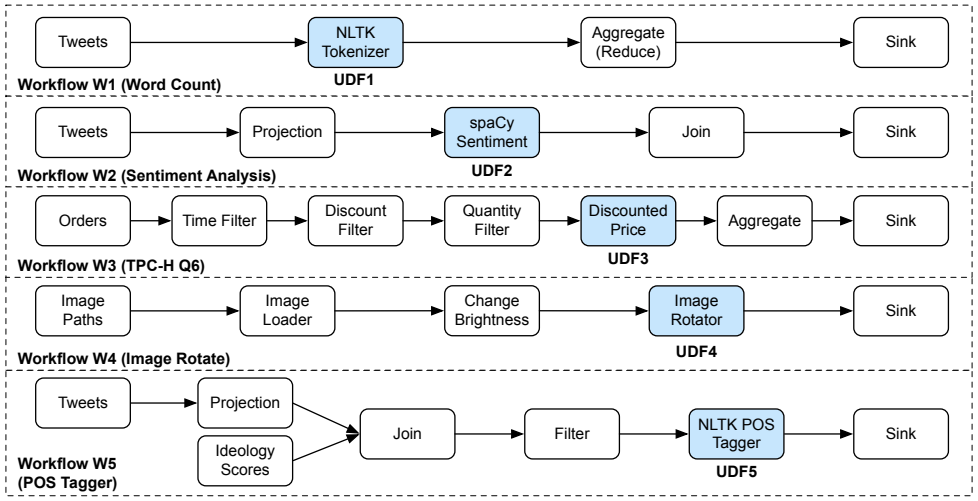

Figure 17. Workflows used in the experiments. We focused on the highlighted UDFs.

Udon Implementation. We implemented Udon on top of the Amber dataflow engine [18] in the Texera system [35]. The Amber engine is implemented in Scala and it supports Python UDF execution using the architecture described in Section 2.1. We chose Amber due to its capability of sending direct control messages between a coordinator and an operator. We extended its control-message channel to deliver debug commands from the coordinator to an operator in PVM, and to deliver debug events in the opposite direction. We used Python's built-in debugger, `pdb` [23], which is widely used in the Python community. We followed the approach described in Section 3.2 to integrate `pdb` with the two-thread execution model. In the UDF code, users optionally add `yield` statements to perform explicit checking of the debug instructions. In all the experiments, Udon was configured to send a `continue` debug command immediately after each received a "breakpoint hit" event to resume the execution. This setting ensures that the debugger can be enabled to monitor the code, and the breakpoints can be hit without pausing the execution.

Machine Environment. For single-machine experiments, we used a virtual machine (VM) on Google Cloud Platform (GCP) with 4 vCPU cores and 32 GB of RAM. For multi-machine experiments, we used a cluster consisting of eight instances of the above VM, providing 32 vCPUs and 256 GB of RAM for the cluster. The VMs were configured to run on the Ubuntu 20.04 operating system and were connected using a high-speed network interface to enable fast communication between nodes. The results were based on Python version 3.10.10.

7.2 Debugging Overhead

In this experiment, we evaluated the debugging overhead of UDFs by comparing their total execution time with and without the debugger. We divided the overhead into the time spent on breakpoint checks and condition evaluations. We added breakpoints to the UDF code with conditions of different types of evaluation complexity. We measured the total execution time of each test using Python's built-in time library and compared the results. To measure the time spent to evaluate conditions, we added timing code into pdb around the line to evaluate the condition of a breakpoint. As measuring the time spent on breakpoint checks conducted by the tracing module was difficult, we measured it by deducting the time spent on condition evaluations from the total overhead. For each workflow, we conducted five test runs with debuggers enabled on the UDF and five runs with debuggers disabled, and reported results in average.

Table 1 shows the total execution time of each UDF compared to the total execution time of the corresponding workflow. When running without a debugger, UDFs already dominated the majority of the execution time in a workflow. UDF_3 accounted for 92.45% of W_3's total time, as the other operators mainly consisted of filters, which were computationally inexpensive. UDF_2 had a higher dominance compared to UDF_3 because sentiment analysis was computationally more expensive than calculating the sum of orders. UDF_5 exhibited the lowest proportion in the total execution time of W_5, primarily due to the presence of a Join operator, which consumed a significant amount of time. UDFs became the bottleneck of a workflow for the following reasons. *i)* Compared to native operators, UDF operators had more complex logic. *ii)* As a UDF required the input data to be transferred from JVM to PVM, and the output tuples to be transferred back to JVM from PVM, it would incur much higher serialization and deserialization cost. *iii)* The Python language is known to be often slower than Java/Scala. With the debugger enabled, the proportion of UDF execution time increased significantly. All UDFs accounted for more than 90% of the workflow's time. This result was expected since native operators did not incur debugging overhead.

Table 1. UDF Execution Time vs Workflow End-to-end Latency

No debugger	UDF_1	UDF_2	UDF_3	UDF_4	UDF_5
UDF execution time (s)	184.24	595.66	178.82	321.43	102.40
Workflow execution time (s)	235.12	626.73	193.58	382.01	151.82
Percentage (UDF/Workflow)	78.31%	95.07%	92.45%	84.21%	67.39%
With debugger	UDF_1	UDF_2	UDF_3	UDF_4	UDF_5
UDF execution time (s)	638.99	979.75	383.68	905.18	500.10
Workflow execution time (s)	695.14	1008.46	398.70	968.45	548.11
Percentage (UDF/Workflow)	91.91%	97.09%	96.13%	93.42%	91.19%

Figure 18 shows the breakdown of each operator's execution time with a debugger. Compared to UDFs without the debugger, the execution time with a debugger was notably longer, ranging from 1.6 times (for UDF_2) to 4.9 times (for UDF_5). Moreover, the different complexities of the Python code contributed to varying overhead for breakpoint checks. For instance, UDF_3's implementation was entirely conducted in the Python UDF without invoking any external libraries, while UDF_4 mainly relied on the Pillow [24] library's implementation, which consisted of both Python and C code. The results showed that the breakpoint checks accounted for 111.34s for UDF_3, which was 61.7% of the UDF's execution time. The breakpoint checks for UDF_4 accounted for 387.67s, a significant portion (120.6%) of the UDF's execution time.

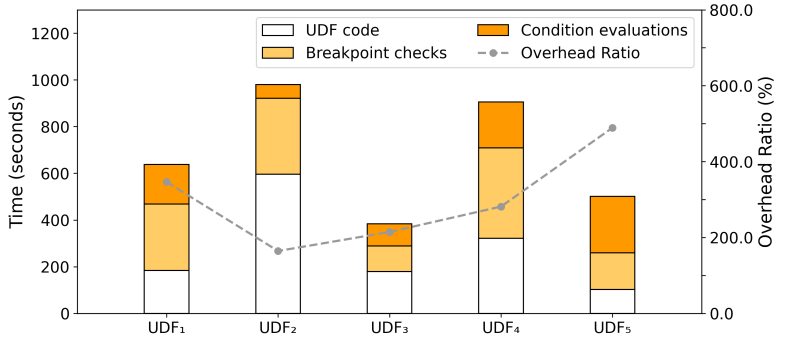

Figure 18. The breakdown of the execution time of UDFs with a debugger.

Additionally, different UDFs exhibited different amounts of overhead, depending on the nature and frequency of breakpoint checks and condition evaluations. For example, we used the same data condition (`"hello" in tuple.text`) for UDF_1 and UDF_2, but their frequencies of breakpoint checks and condition evaluations were different. We added the breakpoint in a for-loop of UDF_1, thus the condition was evaluated repeatedly on each iteration of the loop. Conversely, we added the breakpoint after getting the sentiment of each tweet in UDF_2, resulting in evaluating the same condition only once per tuple. The condition evaluations accounted for 169.24s for UDF_1, which was 91.9% of the UDF's execution time, whereas the condition evaluations only costed 58.67s for UDF_2, which was 9.9% of the UDF's execution time.

7.3 Benefits of Debugging Optimizations

Next, we evaluated the impact of two optimization techniques discussed in Section 5 on the performance of the UDFs under debugging. On top of the same configuration as the experiment in Section 7.2, we tuned the options to enable the two techniques. We considered four execution plans: a) the original plan with full overhead due to the addition of a debugger; b) using optimization technique 1; c) using optimization technique 2; and d) using both techniques.

Figure 19 illustrates how the effectiveness of the optimizations varies across different UDFs. For instance, for UDF_1, optimization 1 reduced the debugging execution time from 638.20s to 384.61s, while optimization 2 reduced it to 363.52s. The optimal execution time of 202.74s was achieved by enabling both optimizations. The results showed that for UDF_1, both optimizations had a positive effect individually and contributed significantly when combined together. As depicted in Figure 18, the debugging overhead of UDF_1 included both large portions of breakpoint checks and condition evaluations and the two optimization techniques were effective in reducing the overhead of each of these overhead portions. In comparison to the baseline of running with simple pdb, Udon successfully reduced the debug overhead ratio of UDF_1 from 246.5% to 10.1%.

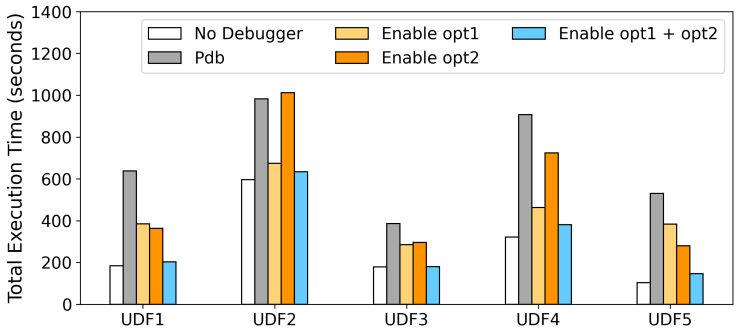

Figure 19. Comparing total execution time of the UDFs under Udon with or without the optimizations.

The trends observed in UDF_1 also exist for UDF_3, UDF_4, and UDF_5. For UDF_3, the breakpoint checks and condition evaluations were more evenly distributed, resulting in a similar execution time acceleration with both optimization 1 and optimization 2. For UDF_4, most of the debugging overhead was attributed to the breakpoint checks, making optimization 1 effective in reducing the execution time from 906.61s to 462.74s. Furthermore, the optimal debug overhead ratio for UDF_4 (18.2%) was slightly higher than that of the other UDFs, indicating that the optimizations were less effective in speeding it up. This can be attributed to UDF_4's use of the Pillow library,

which includes a portion of code written in C. The C code was faster than Python code and but not debuggable using a Python-level debugger. As our proposed optimization techniques target Python code, they are more effective in reducing debugging overhead in Python code than in C code.

UDF_2 exhibited a slightly different pattern compared to other UDFs. While combining the two optimization techniques yielded the optimal execution time, applying optimization 2 separately did not reduce the execution time. From Figure 18, we can see that UDF_2 used very little time for condition evaluations, as the breakpoint condition was checked only once for each tuple. However, optimization 2 introduced more instructions than evaluating the condition to check and pull up the data conditions. As a result, optimization 2 did not reduce the execution time by itself. Nevertheless, when combined with optimization 1, optimization 2 made a small contribution of reducing the execution time from 673.52s to 634.12s, indicating that early evaluation of pulled-up predicates can still be beneficial.

7.4 Scalability of the Optimizations

In this experiment, we evaluated the scalability of Udon with its optimizations using the TPC-H dataset. We conducted the experiment using workflow 3 with two settings. In the first setting, we executed the workflow without any breakpoints or debuggers. In the second setting, we added a breakpoint on UDF_3. For optimal performance, we activated both optimizations within Udon.

We first evaluated the scale-up performance of Udon on a single worker and increased the data size from 200K tuples to 1.6M tuples. As shown in Figure 20, when there were 200K tuples, Udon incurred a 10.3% runtime overhead, with a total execution time of around 173.5s compared to 157.3s without Udon. As we scaled up the input data, the overhead dropped to 8.1%. For larger data scales, the overhead ratio stabilized around 9.0%. This trend occurred because the proportion of initial analysis and optimization techniques required by Udon decreased with an increase in the number of data tuples, and the overhead got normalized with even larger data sizes.

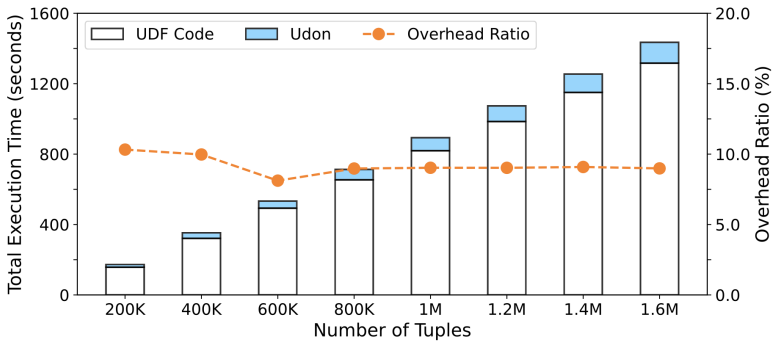

Figure 20. The total execution time of UDF_3 with different input data sizes.

Next, we conducted a scale-out performance evaluation of Udon by testing the workflow with an increasing number of workers from 1 to 8, with the data size also increasing proportionally with worker count from 800K to 6.4M tuples. As shown in Figure 21, for a single worker processing 800K tuples, Udon incurred a runtime overhead of around 9.0%, with a total execution time of around 712.89s compared to 654.17s without Udon. As the number of workers increased to 2, 3, and 4, each processing 800K tuples, the runtime overhead decreased to 8.1%, 6.7%, and 6.6%, respectively. Further increasing the number of workers stabilized the overhead at around 6.1%.

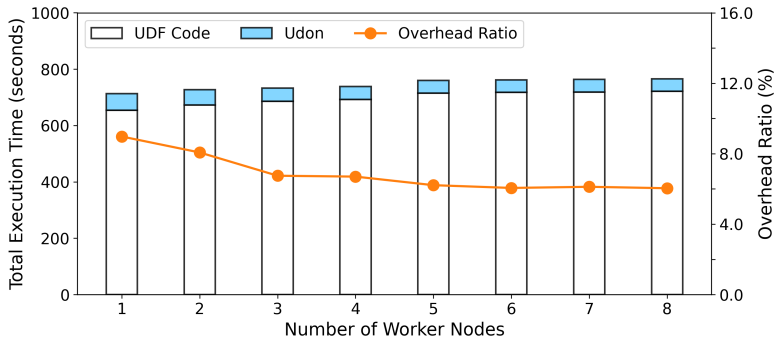

Figure 21. The total execution time of UDF_3 using a different number of parallel workers, each processing the same amount of data tuples.

7.5 Inter-operator Debugging

We evaluated the two state-transfer methods discussed in Section 6 to evaluate conditions between multiple UDF operators. We used the workflow in Figure 22 with UDF_2 (spaCy [32] sentiment analysis) and UDF_5 (NLTK [4] POS Tagger) to conduct the experiment. Each task required a tokenization step, with each UDF using its corresponding library's tokenizer which generate different token sets. To identify and handle differences between token sets generated by spaCy and NLTK tokenizers, we set a breakpoint with a condition that can be triggered when the two libraries produce different tokens. This breakpoint enabled us to pause the execution and examine the difference between the token sets. To handle such a condition, the state (i.e., spaCy's tokens value after line 12) needs to be transferred from the spaCy operator to the NLTK operator.

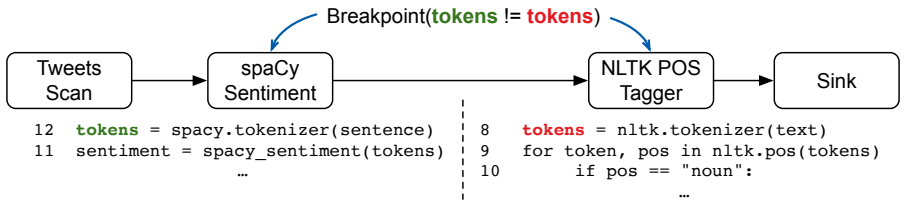

Figure 22. A workflow containing spaCy sentiment and NLTK POS tagger UDFs with different tokenizers.

We implemented both the active and passive state-transfer methods and evaluated their performance. In the active method, we inserted a line `tuple["spacy_tokens"] = tokens` after line 12 of the spaCy operator to append a copy of the state to the tuple. For the passive method, we inserted a line `debugger.states[(tid, lineno, executeseq, tokens)] = tokens` after line 12 of the spaCy operator to store the state in the debugger. In the state, `tid` is the reference id of a tuple and `lineno` is 12. We also have a sequence number `executeseq`, which is used to uniquely identify an execution if it is executed multiple times, such as in a loop. Before line 8 of the NLTK operator, we inserted a line `debugger.request(tid, lineno, executeseq, state)` to request the specific state from the spaCy operator. As shown in Figure 23, without state transfer, the workflow finished in 56.04s. We used a state (token list) with a size of 0.1 KB and transferred it once per tuple using both active and passive methods, resulting in the execution time of 57.42s and 96.28s, respectively. One factor that affected the execution time was the state size. We increased the state size by duplicating the token list (from 1 KB to 10 KB). With a state size of 10 KB, the active method showed

a slight increase in the execution time from 57.42s to 57.92s, and 61.54s. In contrast, the passive method showed a drastic increase in the execution time from 96.28s to 103.19s, and 161.06s.

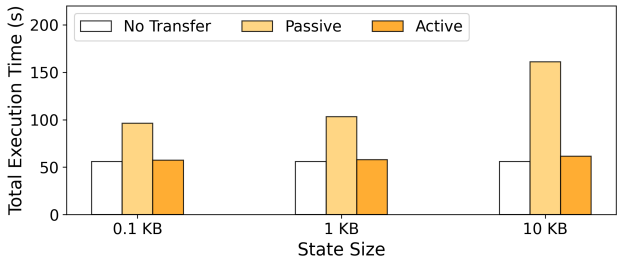

Figure 23. Total execution time for active versus passive state transfer with different state sizes.

The performance difference is due to the following factors. *i)* The active method transferred states as a data field, which can be sent in batches and thus can be optimized using batch-based techniques, such as vectorization and compression. The passive method transferred each state with a separate round of messages. *ii)* The passive method required a message to request the state, which blocked the NLTK operator from processing the next tuple before the response of the message was received. The round-trip network time added significant overhead to the NLTK operator. *iii)* In order to handle the state-request message in a timely manner, the spaCy operator must prioritize the request rather than its own processing. Thus, the frequent message to request for the state reduced the throughput of the spaCy operator.

7.6 Effect of Selectivity on Passive Transfer

We investigated the impact of selectivity on the passive state transfer. Specifically, we added a condition with tunable selectivity prior to the conditional breakpoint used in the previous experiment (Section 7.5), so that only a subset of tuples required state transfer. We varied the ratio of tuples that satisfy the condition from 0.1% to 100% and measured the total execution time of the passive state-transfer method for three different state sizes of 0.1, 1 and 10 KB. The results are shown in Figure 24.

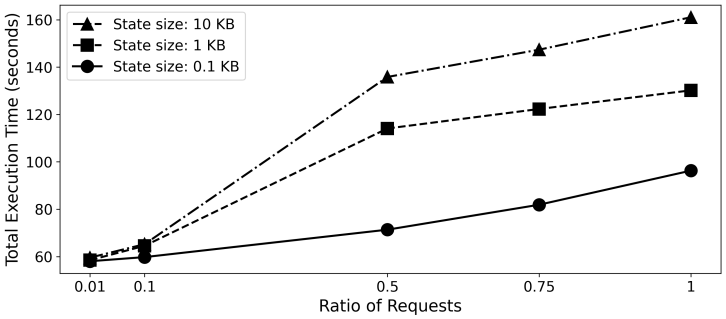

Figure 24. Total execution time of passive state transfer with different selectivities.

As expected, the total execution time increased with an increasing number of tuples satisfying the condition (decreasing selectivity). The time differences were more significant for larger state sizes. This is because the passive approach must initiate state transfers from the upstream operator

for an increasing number of tuples. During each request, the downstream operator was blocked from processing subsequent tuples until the state transfer was completed.

7.7 Effect of Operators on Active State Transfer

We conducted an investigation into the effect of the quantity of operators involved in active state transfer. Since this method relies on transmitting states through data messages along the DAG, the presence of more operators along the path between the sending and receiving operators leads to an increase in the number of state transfers. In order to assess this impact, we introduced multiple dummy operators between the spaCy operator and the NLTK operator, as illustrated in Figure 25.

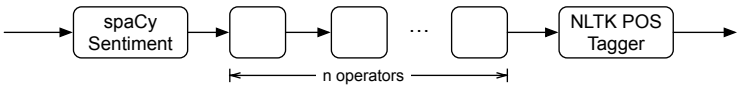

Figure 25. Adding dummy operators in between the sender and receiver UDFs.

We varied the number of dummy operators from 1 to 2, 5, and 10, and compared the total execution time of the workflow with and without the active state transfer method. As shown in Figure 26, the method's execution time demonstrated a clear correlation with the increasing number of dummy operators. This relationship was evident in the increasing message exchange cost between the UDF operators, especially when the state size was large.

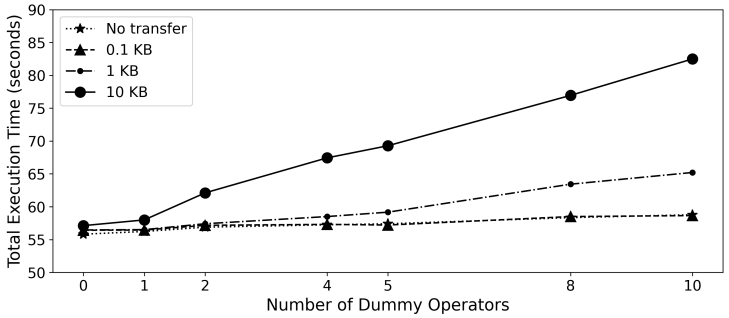

Figure 26. Total execution time for the active state transfer method with different numbers of operators.

8 CONCLUSIONS

In this paper, we proposed Udon, a novel debugger to support fine-grained debugging of UDFs in big data systems. It supports the modern line-by-line debugging primitives such as the ability to set breakpoints, perform code inspections, and make code modifications while executing a UDF on a single tuple. We presented a full specification of Udon, including a novel UDF execution model to ensure the responsiveness of the operator during debugging, advanced state-transfer techniques to satisfy breakpoint conditions across multiple UDFs, and various optimization techniques to reduce the run-time overhead. We conducted a thorough experimental evaluation on multiple datasets and UDF workloads to show its high efficiency and scalability.

ACKNOWLEDGMENTS

We thank Yiming Lin, Xi Lu, Shengquan Ni, the rest of the Texera team at UC Irvine, and the anonymous reviewers for their invaluable feedback. This work was funded by the National Science Foundation (NSF) under award III-2107150.

REFERENCES

[1] Rohan Achar, Pritha Dawn, and Cristina V. Lopes. 2019. GoTcha: an interactive debugger for GoT-based distributed systems. In *Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software, Onward! 2019, Athens, Greece, October 23-24, 2019*, Hidehiko Masuhara and Tomas Petricek (Eds.). ACM, 94–110. https://doi.org/10.1145/3359591.3359733

[2] Apache Hadoop 2023. Apache Hadoop, http://hadoop.apache.org.

[3] bdb 2023. bdb – Debugging framework – Python documentation, https://docs.python.org/3/library/bdb.html.

[4] Steven Bird, Ewan Klein, and Edward Loper. 2009. *Natural Language Processing with Python*. O'Reilly. http://www.oreilly.de/catalog/9780596516499/index.html

[5] Building a non-breaking breakpoint for Python debugging | Opensource.com 2023. https://opensource.com/article/19/8/debug-python.

[6] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. 2015. Apache Flink™: Stream and Batch Processing in a Single Engine. *IEEE Data Eng. Bull.* 38, 4 (2015), 28–38. http://sites.computer.org/debull/A15dec/p28.pdf

[7] Gladys E. Carrillo and Cristina L. Abad. 2017. Inferring Workflows with Job Dependencies from Distributed Processing Systems Logs. In *15th IEEE Intl Conf on Dependable, Autonomic and Secure Computing, 15th Intl Conf on Pervasive Intelligence and Computing, 3rd Intl Conf on Big Data Intelligence and Computing and Cyber Science and Technology Congress, DASC/PiCom/DataCom/CyberSciTech 2017, Orlando, FL, USA, November 6-10, 2017*. IEEE Computer Society, 1025–1030. https://doi.org/10.1109/DASC-PICom-DataCom-CyberSciTec.2017.168

[8] 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/10.1145/3357223.3362738

[9] Darren Dao, Jeannie R. Albrecht, Charles Edwin Killian, and Amin Vahdat. 2009. Live Debugging of Distributed Systems. In *Compiler Construction, 18th International Conference, CC 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings (Lecture Notes in Computer Science, Vol. 5501)*, Oege de Moor and Michael I. Schwartzbach (Eds.). Springer, 94–108. https://doi.org/10.1007/978-3-642-00722-4_8

[10] Debugging | Apache Flink 2023. https://nightlies.apache.org/flink/flink-docs-master/docs/dev/python/debugging/.

[11] Debugging PySpark – PySpark 3.1.1 documantation 2023. https://spark.apache.org/docs/3.1.1/api/python/development/debugging.html.

[12] 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/10.1109/ICDE55515.2023.00276

[13] 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

[14] 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

[15] How does the breakpoint of pdb has affection on performance - StackOverflow 2023. https://stackoverflow.com/questions/73314863/how-does-the-breakpoint-of-pdb-has-affection-on-performance.

[16] Konstantinos Karanasos, Matteo Interlandi, Fotis Psallidas, Rathijit Sen, Kwanghyun Park, Ivan Popivanov, Doris Xin, Supun Nakandala, Subru Krishnan, Markus Weimer, Yuan Yu, Raghu Ramakrishnan, and Carlo Curino. 2020. Extending Relational Query Processing with ML Inference. In *10th Conference on Innovative Data Systems Research, CIDR 2020, Amsterdam, The Netherlands, January 12-15, 2020, Online Proceedings*. www.cidrdb.org. http://cidrdb.org/cidr2020/papers/p24-karanasos-cidr20.pdf

[17] Felix Kossak and Michael Zwick. 2019. ML-PipeDebugger: A Debugging Tool for Data Processing Pipelines. In *Database and Expert Systems Applications - 30th International Conference, DEXA 2019, Linz, Austria, August 26-29, 2019, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 11707)*, Sven Hartmann, Josef Küng, Sharma Chakravarthy, Gabriele Anderst-Kotsis, A Min Tjoa, and Ismail Khalil (Eds.). Springer, 263–272. https://doi.org/10.1007/978-3-030-27618-8_20

[18] Avinash Kumar, Zuozhi Wang, Shengquan Ni, and Chen Li. 2020. Amber: A Debuggable Dataflow System Based on the Actor Model. *Proc. VLDB Endow.* 13, 5 (2020), 740–753. https://doi.org/10.14778/3377369.3377381

[19] Tsung-Yi Lin, Michael Maire, Serge J. Belongie, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollár, and C. Lawrence Zitnick. 2014. Microsoft COCO: Common Objects in Context. In *Computer Vision - ECCV 2014 - 13th European Conference, Zurich, Switzerland, September 6-12, 2014, Proceedings, Part V (Lecture Notes in Computer Science, Vol. 8693)*, David J.

Fleet, Tomás Pajdla, Bernt Schiele, and Tinne Tuytelaars (Eds.). Springer, 740–755. https://doi.org/10.1007/978-3-319-10602-1_48

[20] 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

[21] Barton P. Miller and Jong-Deok Choi. 1988. Breakpoints and Halting in Distributed Programs. In *Proceedings of the 8th International Conference on Distributed Computing Systems, San Jose, California, USA, June 13-17, 1988*. IEEE Computer Society, 316–323. https://doi.org/10.1109/DCS.1988.12532

[22] Christopher Olston, Benjamin C. Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. 2008. Pig latin: a not-so-foreign language for data processing. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2008, Vancouver, BC, Canada, June 10-12, 2008*, Jason Tsong-Li Wang (Ed.). ACM, 1099–1110. https://doi.org/10.1145/1376616.1376726

[23] PDB - The Python Debugger 2023. https://docs.python.org/3/library/pdb.html.

[24] Pillow 2023. Pillow (PIL fork) 10.1.0 documentation, https://pillow.readthedocs.io/en/stable/.

[25] PyDev Debugger 2023. https://www.pydev.org/manual_adv_debugger.html.

[26] PyFlink 2023. PyFlink Docs, https://nightlies.apache.org/flink/flink-docs-master/api/python/.

[27] PySpark 2023. PySpark documantation, https://spark.apache.org/docs/3.1.1/api/python/development/debugging.html.

[28] Python Signal 2023. Signal - Python 3.10.0 documentation, https://docs.python.org/3/library/signal.html.

[29] Mark Raasveldt, Pedro Holanda, Hannes Mühleisen, and Stefan Manegold. 2018. Deep Integration of Machine Learning Into Column Stores. In *Proceedings of the 21st International Conference on Extending Database Technology, EDBT 2018, Vienna, Austria, March 26-29, 2018*, Michael H. Böhlen, Reinhard Pichler, Norman May, Erhard Rahm, Shan-Hung Wu, and Katja Hose (Eds.). OpenProceedings.org, 473–476. https://doi.org/10.5441/002/edbt.2018.50

[30] Ariel Rabkin and Randy H. Katz. 2010. Chukwa: A System for Reliable Large-Scale Log Collection. In *Uncovering the Secrets of System Administration: Proceedings of the 24th Large Installation System Administration Conference, LISA 2010, San Jose, CA, USA, November 7-12, 2010*, Rudi van Drunen (Ed.). USENIX Association. https://www.usenix.org/conference/lisa10/chukwa-system-reliable-large-scale-log-collection

[31] Viktor Rosenfeld, René Müller, Pinar Tözün, and Fatma Özcan. 2017. Processing Java UDFs in a C++ environment. In *Proceedings of the 2017 Symposium on Cloud Computing, SoCC 2017, Santa Clara, CA, USA, September 24-27, 2017*. ACM, 419–431. https://doi.org/10.1145/3127479.3132022

[32] spaCy 2023. spaCy · Industrial-strength Natural Language Processing in Python https://spacy.io.

[33] Leonhard F. Spiegelberg, Rahul Yesantharao, Malte Schwarzkopf, and Tim Kraska. 2021. Tuplex: Data Science in Python at Native Code Speed. In *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*, Guoliang Li, Zhanhuai Li, Stratos Idreos, and Divesh Srivastava (Eds.). ACM, 1718–1731. https://doi.org/10.1145/3448016.3457244

[34] Jiaqi Tan, Xinghao Pan, Soila Kavulya, Rajeev Gandhi, and Priya Narasimhan. 2008. SALSA: Analyzing Logs as StAte Machines. In *First USENIX Workshop on the Analysis of System Logs, WASL 2008, San Diego, CA, USA, December 7, 2008, Proceedings*, Greg Bronevetsky (Ed.). USENIX Association. http://www.usenix.org/events/wasl/tech/full_papers/tan/tan.pdf

[35] Texera 2023. Collaborative Data Analytics Using Workflows, https://github.com/Texera/texera/.

[36] TPC-H 2023. TPC-H Homepage, http://www.tpc.org/tpch/.

[37] What determines debugger run-time performance - StackOverflow 2023. https://stackoverflow.com/questions/9346622/what-determines-debugger-run-time-performance.

[38] Why only main thread can set signal handler in Python - StackOverflow 2023. https://stackoverflow.com/questions/44151888/why-only-main-thread-can-set-signal-handler-in-python.

[39] Doug Woos, Zachary Tatlock, Michael D. Ernst, and Thomas E. Anderson. 2018. A Graphical Interactive Debugger for Distributed Systems. *CoRR* abs/1806.05300 (2018). arXiv:1806.05300 http://arxiv.org/abs/1806.05300

[40] Zhihui Yang, Zuozhi Wang, Yicong Huang, Yao Lu, Chen Li, and X. Sean Wang. 2022. Optimizing Machine Learning Inference Queries with Correlative Proxy Models. *Proc. VLDB Endow.* 15, 10 (2022), 2032–2044. https://www.vldb.org/pvldb/vol15/p2032-yang.pdf

[41] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2010. Spark: Cluster Computing with Working Sets. In *2nd USENIX Workshop on Hot Topics in Cloud Computing, HotCloud'10, Boston, MA, USA, June 22, 2010*, Erich M. Nahum and Dongyan Xu (Eds.). USENIX Association. https://www.usenix.org/conference/hotcloud-10/spark-cluster-computing-working-sets

[42] Yunquan Zhang, Ting Cao, Shigang Li, Xinhui Tian, Liang Yuan, Haipeng Jia, and Athanasios V. Vasilakos. 2016. Parallel Processing Systems for Big Data: A Survey. *Proc. IEEE* 104, 11 (2016), 2114–2136. https://doi.org/10.1109/JPROC.2016.2591592

Received April 2023; accepted August 2023